

USAISEC

US Army Information Systems Engineering Command
Fort Huachuca, AZ 85613-5300

U.S. ARMY INSTITUTE FOR RESEARCH
IN MANAGEMENT INFORMATION,
COMMUNICATIONS, AND COMPUTER SCIENCES

Software Reuse Guidelines

(ASQB-GI-90-015)

April 1990

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

19970521 124

AIRMICS
115 O'Keefe Building
Georgia Institute of Technology
Atlanta, GA 30332-0800



THIS DOCUMENT IS UNCLASSIFIED

PLEASE CHECK THE APPROPRIATE BLOCK BELOW:

-AO # M97-08-4897

☐ _____ copies are being forwarded. Indicate whether Statement A, B, C, D, E, F, or X applies.

☒ DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE: DISTRIBUTION IS UNLIMITED

☐ DISTRIBUTION STATEMENT B:
DISTRIBUTION AUTHORIZED TO U.S. GOVERNMENT AGENCIES ONLY; (Indicate Reason and Date). OTHER REQUESTS FOR THIS DOCUMENT SHALL BE REFERRED TO (Indicate Controlling DoD Office).

☐ DISTRIBUTION STATEMENT C:
DISTRIBUTION AUTHORIZED TO U.S. GOVERNMENT AGENCIES AND THEIR CONTRACTORS; (Indicate Reason and Date). OTHER REQUESTS FOR THIS DOCUMENT SHALL BE REFERRED TO (Indicate Controlling DoD Office).

☐ DISTRIBUTION STATEMENT D:
DISTRIBUTION AUTHORIZED TO DoD AND U.S. DoD CONTRACTORS ONLY; (Indicate Reason and Date). OTHER REQUESTS SHALL BE REFERRED TO (Indicate Controlling DoD Office).

☐ DISTRIBUTION STATEMENT E:
DISTRIBUTION AUTHORIZED TO DoD COMPONENTS ONLY; (Indicate Reason and Date). OTHER REQUESTS SHALL BE REFERRED TO (Indicate Controlling DoD Office).

☐ DISTRIBUTION STATEMENT F:
FURTHER DISSEMINATION ONLY AS DIRECTED BY (Indicate Controlling DoD Office and Date) or HIGHER DoD AUTHORITY.

☐ DISTRIBUTION STATEMENT X:
DISTRIBUTION AUTHORIZED TO U.S. GOVERNMENT AGENCIES AND PRIVATE INDIVIDUALS OR ENTERPRISES ELIGIBLE TO OBTAIN EXPORT-CONTROLLED TECHNICAL DATA IN ACCORDANCE WITH DoD DIRECTIVE 5230.25, WITHHOLDING OF UNCLASSIFIED TECHNICAL DATA FROM PUBLIC DISCLOSURE, 6 Nov 1984 (Indicate date of determination). CONTROLLING DoD OFFICE IS (Indicate Controlling DoD Office).

☐ This document was previously forwarded to DTIC on _____ (date) and the AD number is _____.

☐ In accordance with provisions of DoD instructions, the document requested is not supplied because:

☐ It will be published at a later date. (Enter approximate date, if known).

☐ Other. (Give Reason)

DoD Directive 5230.24, "Distribution Statements on Technical Documents," 18 Mar 87, contains seven distribution statements, as described briefly above. Technical Documents must be assigned distribution statements.

VICTOR LA CHANCE
Authorized Signature/Date

Joyce J. Chiras
Print or Type Name
DTIC - OCA
DTIC/BRN
Telephone Number
Per phone conv.
5-16-97
DSN- 478-2413

K/DSRD-150

Data Systems Research and Development Division

SOFTWARE REUSE GUIDELINES

James W. Hooper
University of Alabama in Huntsville
Rowena O. Chester

DATE PUBLISHED — December 1989

Prepared for
U. S. ARMY INSTITUTE FOR RESEARCH
IN MANAGEMENT INFORMATION, COMMUNICATIONS,
AND COMPUTER SCIENCES
INFORMATION SYSTEMS ENGINEERING COMMAND

by

MARTIN MARIETTA ENERGY SYSTEMS, INC.
operating the
Oak Ridge National Laboratory Oak Ridge Y-12 Plant
Oak Ridge Gaseous Diffusion Plant Paducah Gaseous Diffusion Plant
for the
U.S. DEPARTMENT OF ENERGY
under contract DE-AC05-84OR21400

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE


REPORT DOCUMENTATION PAGE

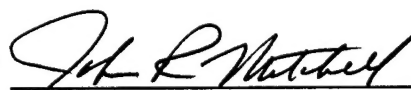
Form Approved
OMB No. 0704--188
Exp. Date: Jun 30, 1986

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS NONE		
2a. SECURITY CLASSIFICATION AUTHORITY N/A			3. DISTRIBUTION / AVAILABILITY OF REPORT N/A		
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE N/A					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) ASQB-GI-90-015			5. MONITORING ORGANIZATION REPORT NUMBER(S) N/A		
6a. NAME OF PERFORMING ORGANIZATION AIRMICS		6b. OFFICE SYMBOL (if applicable) ASQB-GI	7a. NAME OF MONITORING ORGANIZATION N/A		
6c. ADDRESS (City, State, and ZIP Code) 115 O'Keefe Bldg., Georgia Institute of Technology Atlanta, GA 30332-0800			7b. ADDRESS (City, State, and Zip Code) N/A		
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AIRMICS		8b. OFFICE SYMBOL (if applicable) ASQB-GI	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER		
8c. ADDRESS (City, State, and ZIP Code) 115 O'Keefe Bldg., Georgia Institute of Technology Atlanta, GA 30332-0800			10. SOURCE OF FUNDING NUMBERS		
			PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.
			62783A	DY10	02-02-01
11. TITLE (Include Security Classification) Software Reuse Guidelines <div style="text-align: right;">(UNCLASSIFIED)</div>					
12. PERSONAL AUTHOR(S) James W. Hooper; Rowena O. Chester					
13a. TYPE OF REPORT		13b. TIME COVERED FROM _____ TO _____		14. DATE OF REPORT (Year, Month, Day) 1989, December, 13	
				15. PAGE COUNT 71 body + 8	
16. SUPPLEMENTARY NOTATION					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP	Reuse, Metrics, Ada, STARS, RAPID, DOD-STD-2167A, Software Engineering, Maintenance, Domain Analysis, Requirements Specification, Detail Design, Integration Testing, Unit Testing, Detail Design		
19. ABSTRACT (Continue on reverse if necessary and identify by block number) Guidelines for U.S. Army development groups undertaking software reuse are provided, and reuse-related issues discussed pertaining to DOD and other government contracting and funding approaches. The material is also pertinent to any organization engaged in the development and maintenance of large software systems. Problems in software productivity and quality are discussed along with what reuse can achieve. Reuse research and practices are summarized. Managerial and technical reuse issues are discussed including existing impediments to reuse and their resolutions and the creation of incentives for reuse. Instituting a software development and maintenance process incorporating reuse is emphasized. Discussion is provided in specific technical issues of domain analysis, preparing reusable components (spanning the entire software life cycle), assuring component quality, and classifying and storing components. Specific guidelines are provided for developing Ada code modules for reuse. Operational issues in software reuse are considered, including searching, retrieving, understanding, assessing, adapting, and assembling components. Tools and environment for software reuse are also discussed. A phased approach is suggested for implementing a software reuse program. This should help to keep risk manageable, develop necessary skills, and demonstrate the viability of reuse.					
20. DISTRIBUTION / AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED / UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a. NAME OF RESPONSIBLE INDIVIDUAL Howard C "Butch" Higley			22b. TELEPHONE (Include Area Code) (404) 894-3110		22c. OFFICE SYMBOL ASQB-GI

This research was performed as an in-house project at the Army Institute for Research in Management Information, Communications, and Computer Sciences (AIRMICS), the RDTE organization of the U.S. Army Information Systems Engineering Command (USAISEC). This effort was performed under the AIRMICS Technology Insertion Program to support the U.S. Army Information Systems Command (USAISC) in the development of a report entitled "Long Range Planning Guidance - Objective Configuration." An initial meeting was held in early December in Atlanta to coordinate the task. Twenty-six topics were selected for consideration, with AIRMICS agreeing to conduct technology assessments on fifteen of the topics. Planning Research Corporation (PRC) was assigned responsibility for conducting the remaining assessments and consolidating all the assessments for use in the planning document. In a two-week period, AIRMICS completed the assessments and provided the results to ISC-DCSPLANS and ISEC-SID. This research report is not to be construed as an official Army position, unless so designated by other authorized documents. Material included herein is approved for public release, distribution unlimited. Not protected by copyright laws.

THIS REPORT HAS BEEN REVIEWED AND IS APPROVED

s/ 
Glenn E. Racine
Chief
CISD

s/ 
John R. Mitchell
Director
AIRMICS

CONTENTS

PREFACE	v
EXECUTIVE SUMMARY	vii
ABSTRACT	ix
1. BACKGROUND AND INTRODUCTION	1
1.1 THE PROBLEM	1
1.2 CONCEPTS AND DEFINITIONS	1
1.3 RESEARCH ACTIVITIES	3
1.4 STATUS OF REUSE PRACTICE	4
1.5 SCOPE AND ORGANIZATION OF THIS DOCUMENT	6
2. MANAGERIAL GUIDELINES	7
2.1 MANAGERIAL ISSUES AND APPROACHES	7
2.1.1 Leadership and Organizational Structure	7
2.1.2 Organizational Behavior	8
2.1.3 Contractual and Legal Considerations	9
2.1.4 Financial Considerations	11
2.2 SOFTWARE DEVELOPMENT AND MAINTENANCE INCORPORATING REUSE	12
2.2.1 Software Process Concepts and Issues	12
2.2.2 A Generic Reuse/Reusability Model	13
2.2.3 Alternatives to the Waterfall Model	15
2.2.4 Establishing a Process	16
3. TECHNICAL GUIDELINES	17
3.1 DOMAIN ANALYSIS	17
3.2 CREATING REUSABLE COMPONENTS	21
3.2.1 Reusable Components Spanning the Life Cycle	21
3.2.2 Code Components	24
3.2.2.1 Passive Versus Dynamic Code Components	24
3.2.2.2 Programming Style	24
3.2.3 Guidelines for Writing Reusable Ada Code	26
3.2.4 Component Quality	31
3.2.5 Classifying and Storing Components	34
3.3 REUSING COMPONENTS	37
3.3.1 Searching and Retrieving	37

3.3.2 Understanding and Assessing Components	39
3.3.3 Adapting Components	40
3.3.4 Composition of Code Components	42
3.4 TOOLS AND ENVIRONMENTS	43
4. GETTING STARTED	47
REFERENCES	49
APPENDIX A: COLLECTED GUIDELINES	59
ACRONYMS	71

PREFACE

This document is intended to provide guidance to U.S. Army development groups undertaking software reuse. Some issues are discussed that are of specific concern to the U.S. Department of Defense (DOD) and other government agencies, including contracting and project funding issues. However, most of the material is relevant to any organization developing and maintaining large software systems. This document is one of the contributions of the Ada Reuse and Metrics project conducted for the Army Institute for Research in Management Information, Communications, and Computer Sciences (AIRMICS) by Data Systems Research and Development of Martin Marietta Energy Systems, Inc., Oak Ridge, Tennessee, under Contract No. DE-AC05-84OR21400. Energy Systems is a prime operating contractor for the U.S. Department of Energy (DOE). The Reuse and Metrics project was funded by the Software Technology for Adaptable, Reliable Systems (STARS) program, and addressed issues affecting reuse throughout the software engineering life cycle. Energy Systems subcontracted with a number of universities to carry out research addressing different aspects of reuse. The universities, the principal investigators, and areas of emphasis were:

George Mason University and CONTEL; Richard E. Fairley, Shari Lawrence Pfleeger; incentives for software reuse, economics of software reuse.

Georgia State University; Ross A. Gagliano; reusable Ada library tools.

Georgia Institute of Technology; W. Michael McCracken; verification and validation of reusable Ada components.

Morehouse College; Arthur Jones; framework for library and configuration management.

Purdue University; Richard A. DeMillo, Rhonda Martin; reusing operational history of Ada software components, DOD standards harmonization.

University of Alabama in Huntsville; James W. Hooper; formulation of reuse guidelines.

University of Colorado at Boulder; Roger King; object-oriented database modeling and software environments.

University of Maryland (College Park); Victor R. Basili, H. Dieter Rombach; Ada reuse metrics, Ada data binding tool.

University of Houston at Clear Lake; Charles W. McKay; conceptual and implementation models of the software life cycle.

Daniel E. Hocking served as the AIRMICS Project Manager; Rowena O. Chester (and earlier, Peter A. Lesslie) served as the Energy Systems Project Manager.

The body of this document includes material summarized from the Ada Reuse and Metrics project, some of which appears in *Guidelines Document for Ada Reuse and Metrics (Draft)* (Lesslie, Chester, and Theofanos 1989). Individual papers describing the research will be published separately from this document. Reuse ideas and guidelines for this document were drawn from a great many sources in addition to the research conducted for AIRMICS, as is evident from the extensive References section. In fact, by volume the description of the AIRMICS research occupies a relatively small part of the document. We have sought to include an

overview of many current approaches to reuse issues in order to familiarize the reader with multiple viewpoints as well as to instill a better appreciation for the range of possible approaches and aspects of reuse.

EXECUTIVE SUMMARY

Research and experimentation in software reuse have been conducted to help alleviate the critical problems of low software productivity and poor software quality. Sufficient progress has now been made that specific recommendations can be made for integrating reuse into the process of software development and maintenance. Availability of the Ada language has served as a strong motivator for reuse and as a primary mechanism for reusability.

Software reuse has been characterized as an act of synthesizing a solution to a problem based on predefined solutions to subproblems. This definition emphasizes the reuse of predeveloped solutions across the software's entire life cycle—with the greatest leverage expected to come from the reuse of products from earlier life-cycle phases. The implications of this definition include recognizing the desirability of retaining certain predefined solutions for reuse, codifying and retaining the solutions, recognizing the availability of potentially applicable components during solution of a problem, and adapting and composing components into software that provides a valid problem solution.

Considerable research and experimentation have now been achieved in all these facets of reuse. Additional important research is underway at the Software Engineering Institute (SEI), the Software Productivity Consortium (SPC), the Microelectronics and Computer Technology Corporation (MCC), government and industrial laboratories, and universities. Now in place are such support facilities as the DOD Ada Software Repository and AdaNET. The Information Systems Engineering Command (ISEC) is engaged in an 18-month pilot project that is experimenting with reusable Ada packages for information systems development (RAPID). Numerous corporations have achieved considerable success in reuse across a wide spectrum of applications.

Strong, high-level leadership must be provided to make reuse a reality. This involves committing necessary resources (for software engineering process, tools, staff, and library), bringing about any necessary changes to the organization, establishing realistic goals and accepting risks, putting into place rewards for good performance in reuse (for managers and technical personnel), and making personnel assignments that emphasize reuse. There also are existing individual behavior patterns that must be overcome. Existing contractual, legal, and project funding approaches need careful consideration.

An extremely important need, with broad implications, is to devise software engineering processes that support and encourage reuse. While DOD-STD-2167A permits reuse, it does not actively support and encourage it. Thus the following steps are suggested for use at each life-cycle phase:

1. understanding the problem and identifying a solution structure based on the predefined components;
2. reconfiguring the solution structure to improve the possibility of using predefined components available at the next phase;
3. acquiring, assessing, instantiating, and modifying predefined components;
4. integrating the components into the products for this phase; and
5. evaluating reusability prospects of components that must be developed and of components obtained by modifying predefined components.

Successful reuse projects have demonstrated that, even without major technology breakthroughs, significant benefits are possible from software reuse. No doubt,

better, yet to be devised, solutions to some of the technical problems can provide much greater leverage than is now possible. Domain analysis is a relatively new endeavor in which certain carefully bounded application areas, for which reuse seems promising, are selected and the objects, operations, and relationships of each domain are characterized in the form of a "domain model." A dictionary of domain terminology and a "domain architecture" to support and to guide the synthesis of systems from the reusable components are devised. This analysis can guide decision making about the development of individual reusable components. It costs extra money to prepare for reusability (due to the need to generalize the applicability, to perform needed rigorous verification and validation, and to catalog and retain for use). The domain analysis also can provide guidance when reuse of a component is contemplated.

This document discusses reuse and guidelines relative to the preparation of reusable components, including interrelationships of components spanning the life cycle, and suggested steps to follow at each life-cycle phase. Detailed suggestions are offered for the creation of code modules, including the different forms code modules can take, programming style considerations, and guidelines for the use of Ada in preparing code for reuse. Component quality and mechanisms for classifying and storing components also are emphasized.

Guidelines are offered for reusing components, including the issues of searching and retrieving, understanding and assessing candidate components, adapting components for use, and assembling reusable components into executable software. Consideration is given to tools and environments for reuse. A complete list of the guidelines, both managerial and technical, is provided in Appendix A.

A phased approach is suggested for implementing a software reuse program. This approach should help to keep risk manageable, develop necessary skills and demonstrate the viability of reuse. The following list of steps summarizes the suggested phased approach.

1. Assess software engineering practices, remedy shortcomings.
2. Obtain top-level management support for software reuse.
3. Identify one or more candidate application areas for reuse consideration.
4. Inventory reusable assets for the identified applications area(s).
5. Establish an initial library of reusable components.
6. Conduct a pilot project.
7. Evaluate experience/success and obtain management decision to proceed or not.
8. As success and management approval warrant, expand reuse to other domains and organizational segments.
9. Conduct the following activities as part of the practice of software reuse:
 - * Encourage reuse by management policies and practices.
 - * Make reuse an integral part of the software engineering process.
 - * Adjust organization and staffing as necessary.
 - * Provide adequate library mechanisms.
 - * Perform domain analysis of selected domain(s) and provide reusable components for the domain(s).
 - * Continually assess effectiveness and adjust/augment as appropriate.

ABSTRACT

Guidelines for U.S. Army development groups undertaking software reuse are provided and reuse-related issues discussed pertaining to DOD and other government contracting and funding approaches. The material is also pertinent to any organization engaged in the development and maintenance of large software systems.

Problems in software productivity and quality are discussed along with what reuse can achieve. Reuse research and practices are summarized.

Managerial and technical reuse issues are discussed including existing impediments to reuse and their resolutions and the creation of incentives for reuse. Instituting a software development and maintenance process incorporating reuse is emphasized.

Discussion is provided on specific technical issues of domain analysis, preparing reusable components (spanning the entire software life cycle), assuring component quality, and classifying and storing components. Specific guidelines are provided for developing Ada code modules for reuse. Operational issues in software reuse are considered, including searching, retrieving, understanding, assessing, adapting, and assembling components. Tools and environments for software reuse are also discussed.

A phased approach is suggested for implementing a software reuse program. This should help to keep risk manageable, develop necessary skills, and demonstrate the viability of reuse.

1. BACKGROUND AND INTRODUCTION

1.1 THE PROBLEM

In 1980 DOD spent over \$3 billion on software; by 1990 DOD software expenses were predicted by Horowitz and Munson (1984) to exceed \$30 billion. Even though expenditures are escalating, productivity is falling behind the demand for new software. Delivered software is often of poor quality and very difficult (and expensive) to maintain. The same trends are perceivable throughout the software industry. Software reuse, however, has the potential to increase productivity, reduce costs, and improve software quality. Jones (1984) estimates that of all the code written in 1983, probably less than 15 percent is unique, novel, and specific to individual applications. Thus an obvious candidate area for increasing productivity and reducing cost is to reuse existing software products to achieve all or part of the redundant 85 percent of the development. An estimated average of only about five percent of code is actually reused (Frakes and Nejme 1987, quoting DeMarco). Even a one percent gain, relative to DOD's projected \$30 billion, could save \$300 million. Software quality improvements are expected to come from the greater use and testing of individual components, with the resulting isolation and correction of any problems discovered.

Some software reuse has occurred for many years, of course, beginning with libraries of mathematical subroutines and now including operating systems, language processors, report generators, compiler generators, fourth-generation languages, and many application-specific packages. To achieve the needed benefits, however, software reuse must be expanded much further—to "reuse in the large"—to cope with the size and complexity of current software.

Software reuse cannot be successful in isolation; it must be applied within a framework of effective software engineering practice including organizational structure, life-cycle process (policies, methods, tools, etc.), and well-trained personnel.

1.2 CONCEPTS AND DEFINITIONS

The candidate products for reuse may be considered and characterized relative to activities of the software development and maintenance process. We can characterize these activities as:

- * domain analysis
- * requirements specification
- * high-level design
- * detailed design
- * coding and unit testing
- * integration testing
- * documentation
- * maintenance

In principal, we should expect a greater return from the higher-level abstraction activities, if their products can be reused. Thus, a reused requirements specification should give us greater leverage than a reused code module. Only part of the knowledge gained from the software life-cycle activities is usually recorded and retained.

2 SOFTWARE REUSE GUIDELINES

In addition to requirements specifications, designs, code, test documents, test cases, integration plan, etc., which are recorded, lessons learned about the application domain during a project and rationale for design decisions, tradeoff considerations in decomposition and allocation to system components, etc. should be recorded. Almost always some knowledge is factored out through the refinement process; but knowledge acquired during a project is exceedingly valuable for retention.

It should be noted that experienced personnel naturally retain much knowledge of their previous work; thus reuse of personnel is extremely advantageous in software development within an application area. Also, the software development/maintenance process(es) used within an organization are reused to great advantage—personnel learn how to approach problem solutions in a systematic way, and they benefit from repetitive application of the process. In this document, the term software component (or component) is used to mean any type of software resource that may be reused (e.g., code modules, designs, requirements specifications, or domain knowledge).

The way we define terms is critical, since our understanding is determined (focused or limited) by our definitions. Two fundamental definitions of software reusability are:

1. the extent to which a software component can be used (with or without adaptation) in a problem solution other than the one for which it was originally developed; and
2. the extent to which a software component can be used (with or without adaptation) in multiple problem solutions.

Definition 1 suggests that reuse is an incidental result from software development; definition 2 suggests planned efforts toward reuse. While the prefix "re" of the word reusability may inherently suggest definition 1, definition 2 appears to be a more productive and better-focused definition and likely better characterizes current thinking. Perhaps definition 1 better characterizes the previous achievements in reuse, and definition 2 recognizes the need to emphasize reuse as a worthy focus within itself. Clearly, reuse is the goal, while reusability is necessary in order to achieve the goal. A good definition of reuse (Kang 1987), consistent with definition 2 is:

Reuse is an act of synthesizing a solution to a problem based on predefined solutions to subproblems.

There are numerous implications of this definition, including recognizing the desirability of retaining certain predefined solutions for reuse, codifying and retaining the solutions (i.e., components), recognizing the availability of potentially applicable components during solution of a problem, and adapting and composing the components into software that provides a valid problem solution. To achieve software reuse, means must be provided to deal with these implications. The focus of this document is to consider various means devised to deal with the issues and to offer guidelines for instituting software reuse within an organization (especially a government organization).

Portability is a characteristic of software closely related to reusability. Portability refers to the extent to which a software component can be used in multiple machine environments (physical hardware, operating system, run-time environment, and compiler conventions). Thus reusability includes portability in the sense that portability is necessary to achieve reusability across multiple machine environments. Section 3.2.4 (Component Quality) considers the interrelationships between portability and adaptability.

1.3 RESEARCH ACTIVITIES

Both in the U.S. and abroad (especially in Europe and Japan) a great deal of research is underway to achieve effective software reuse. Although it is not our purpose to present detailed coverage of ongoing research projects, it is useful to present an indication of the extent of reuse research activity as a measure of the importance currently placed on software reuse.

The Common Ada Missile Packages (CAMP) project (Anderson and McNicholl 1985; McNicholl et al. 1986), conducted by McDonnell Douglas under contract to the DOD STARS program, has proven to be one of the most important reuse research projects conducted. The federally funded SEI at Carnegie-Mellon University in Pittsburgh is conducting the Applications of Reusable Software Components (ARSC) project, which is experimenting with the CAMP reusable parts by developing software for the Tomahawk missile system (Holibaugh 1989). The work emphasizes various facets of reuse, including domain analysis and domain-specific software architectures. The U.S. Army Communications Electronics Command (CECOM) Center for Software Engineering also is conducting research based on the CAMP parts. STARS efforts also resulted in the development of a reusability guidebook (Wald 1986), and STARS is now undertaking a major reuse research project involving support from Boeing, IBM, and Unisys.

U.S. Army AIRMICS has sponsored reuse research at a number of universities and institutions, leading to generation of this document. Martin Marietta Energy Systems directed the research, and the STARS program was the funding source. The RAPID Center project is being conducted by SofTech for the U.S. Army ISEC; it emphasizes the identification and retrieval of reusable Ada software components (Guerrieri 1988; Vogelsong 1989). ISEC also sponsored SofTech's work in preparing guidelines for reuse (ISEC 1985). SofTech also prepared guidelines for the U.S. Air Force Electronic Systems Division (Braun, Goodenough, and Eaves 1985). Honeywell prepared guidelines for reusable Ada code under funding from Office of Naval Research (St. Dennis 1986). U.S. Air Force Rome Air Development Center (RADC) has sponsored reuse research (Presson et al. 1983).

The SPC in Reston, Virginia, conducts reuse research, including studies involving cost modeling for reuse assessment and relationships between reuse and prototyping (Pyster and Barnes 1987; Barnes et al. 1987a, 1987b). MCC in Austin, Texas, is conducting research in many facets of reuse, including the application of reverse engineering methods and hypermedia to reuse, and reuse of software components across the life cycle, which is called wide-spectrum reuse (Biggerstaff and Richter 1987; Biggerstaff 1989; Lubars 1987).

Numerous companies are active in reuse research and experimentation. In addition to those already mentioned, some others are: Computer Sciences Corporation, Computer Technology Associates, CONTEL Technology Center, Draper Labs, GTE, Institute for Defense Analyses, Rational, SAIC, and Westinghouse. A summary of some reuse projects as of 1987 may be found in Tracz 1988b.

Significant research projects are underway in the United Kingdom, as evidenced by the special section on software reuse in the September 1988 issue of the *Software Engineering Journal* (Hall 1988); this issue contains some excellent research papers. The European Software Factory (ESF) is a multi-nation project to advance reuse knowledge/practice in Europe. A recent paper on the Japanese "software factory" approach is Fujino 1987.

Many conferences and workshops dealing with software reuse are being held, and refereed journal articles on reuse are becoming more numerous. The nature of the papers shows that research issues are becoming more focused, and progress is

4 SOFTWARE REUSE GUIDELINES

evident. The References Section includes many papers on research results. A seminal reference is the landmark September 1984 issue on software reusability of the *IEEE Transactions on Software Engineering* (Biggerstaff and Perlis 1984). Other important references are the July 1987 special issue of *IEEE Software*, "Making Reuse a Reality" (Tracz 1987b). An outstanding paper is Biggerstaff and Richter 1987, and Hooper (1989) provides an overview of software reuse research and practice. Four tutorial volumes are available: Freeman 1987, Tracz 1988a, and Biggerstaff and Perlis 1989a, 1989b.

1.4 STATUS OF REUSE PRACTICE

Reuse concepts are moving from the research stage into practice, and some very good results are being reported. An initial investment to undertake reuse (organizational changes, initial library development, training, etc.) is required, and there has been an understandable reluctance to make this investment without a reasonable assurance of success. Enough reuse success stories are accumulating to begin to allay the concerns; thus we should expect an increase in the number of organizations undertaking the practice of software reuse. A number of the successes have been based on ad hoc approaches, which means that technical breakthroughs are not necessary to achieve success in software reuse, although productivity can certainly be further improved. We will consider in this section a number of success stories as well as some mechanisms now available to support reuse.

The DOD Ada Software Repository and AdaNET are reuse support mechanisms now in place. The DOD Ada Software Repository was established in 1984 to promote the exchange and use of public-domain Ada programs and tools, and to promote Ada education by providing several working examples of programs in source code form to study and to modify. The repository contains source code exceeding 20 MB in size. Conn (1986) provides an overview of the DOD Ada Software Repository and explains how to obtain access to available services. AdaNET is a government-sponsored information service, established in October 1988 to facilitate the transfer of federally developed software engineering and Ada technology to the private sector. It is operated by MountainNet, Inc., Dellslow, West Virginia, and sponsored by the National Aeronautics and Space Administration (NASA) Technology Utilization Division, DOD Ada Joint Program Office (AJPO), and Department of Commerce Office of Productivity, Technology, and Innovation (OPTI). AdaNET offers 24-hour online computer access to information about Ada software, bibliographies, conferences and seminars, education and training, news events, products, reference materials, and standards. Interested organizations and individuals are invited by MountainNet to call them for detailed information, including how to apply for an AdaNET Electronic Mail Account. (The telephone number is 304-296-1458.) Both of these software reuse support mechanisms, Ada Software Repository and AdaNET, have the potential to provide practical help to organizations practicing reuse, although their use to date has been limited.

A number of documents are available that offer guidance in practicing software reuse—primarily in preparing Ada code for reuse. These include: ISEC 1985; Braun, Goodenough, and Eaves 1985; Wald 1986; and St. Dennis 1986. This document has drawn from these guides, and covers a wider range of reuse issues. Booch (1987) is an excellent reference on the use of Ada in creating reusable software components; he emphasizes object-oriented design (OOD). EVB Software Engineering, Inc., markets a set of Ada software modules based on Booch's work (including components for stacks, lists, strings, queues, dequeues, rings, maps, sets, bags, trees, graphs, filters, pipes, sorting, searching, and pattern matching).

Vogelsong (1989) describes the status of the RAPID project. The goal of RAPID is to promote the reuse of Ada software, and to reduce the cost of system development and maintenance. Developed by SofTech, under contract to ISEC, RAPID became operational May 1, 1989, for an 18-month pilot operational period (at the U.S. Army Information Systems Software Development Center in Falls Church, Virginia). During the first 9 months a single development effort will be supported to prove reusability concepts, refine library software, and resolve contractual and management issues. During the remaining 9 months, the feasibility of servicing five Software Development Centers will be assessed. The long-range plan is to include all of ISEC, Department of the Army, etc., as need and funding allow.

The initial RAPID domain analysis covered information management systems (financial, logistical, tactical management information, communication, personnel/force accounting, etc.), but the analysis is expected to expand to additional domains and to accommodate multiple projects. The RAPID Center Library (RCL) operates on a MicroVAX II and consists of 30,000 lines of Ada code. RAPID has a support staff to provide consultation on reuse throughout a project's life cycle. ISEC's experience with RAPID should be very valuable to other DOD segments as well as to other government and commercial organizations and universities.

An example of a successful reuse project is reported by Carstensen (1987) of Magnavox. The AFATDS project for the U.S. Army consisted of approximately 770,000 lines of Ada code, of which about 100,000 lines were reused code. Of the 100,000 lines, about 30,000 were reused unchanged and about 70,000 resulted from tailoring existing modules. Magnavox used OOD which facilitated reuse. By way of incentive for reuse, at project initiation they determined and costed a specific software reuse factor that had to be met to stay on schedule and within budget. As Carstensen notes, this required acceptance of some risk by project-level management, and he emphasizes that whatever incentives are used, the single most important incentive (factor) is the acceptance by project management of any real or perceived risks associated with the reuse of previously developed software.

Selby (1989) studied software reuse activities at NASA Goddard Space Flight Center (GSFC). GSFC has active research projects in reuse, and practices reuse. Selby considered 25 moderate and large-size software systems (from 3,000 to 112,000 lines of Fortran source code) that are support software for unmanned spacecraft control. The amount of software either reused or modified from previous systems averaged 32 percent per project. Subsequent efforts at GSFC with Ada code indicate reuse averages even higher than the 32 percent level experienced with Fortran code.

Lanergan and Grasso (1984) also emphasize the importance of management commitment in Raytheon's successful reuse project. The Information Processing Systems Organization of Raytheon's Missile Systems Division concluded that about 60 percent of their business application designs and code were redundant. By standardizing those functions in the form of reusable functional modules and logic structures, they are experiencing about a 50 percent gain in productivity. Also, they report marked improvement in the maintenance process due to a consistent style for all software, which permits the reassignment of personnel from maintenance to development of new systems.

Biggerstaff and Perlis (1989b) reprint the Lanergan and Grasso (1984) paper, as well as papers by Prywes and Lock (1989) and Cavaliere (1989) on reuse in business applications. Prywes and Lock used a program generator approach, with results of a threefold gain in programmer productivity. Biggerstaff and Perlis observe that, while Cavaliere reports on an ad hoc approach at ITT Hartford Insurance, good results were obtained, largely because "the Hartford management supported,

6 SOFTWARE REUSE GUIDELINES

capitalized, and actively moved to assure the success of the project." Biggerstaff and Perlis also reprint and comment on papers by Oskarsson (1989) and Matsumoto (1989), reporting reuse successes in telephony software and process control software, respectively. They note that reuse skeptics have doubted the possibilities of reuse in these domains, since these domains impose unusually strict memory requirements and performance constraints. Biggerstaff and Perlis note that these experiences reflect "what can be accomplished with enlightened and committed management coupled with existing technology." They suggest that a manager who is considering undertaking reuse would do well to read about the experiences of these organizations (Biggerstaff and Perlis 1989b, pp. xviii-xx).

Tracz (1987c) believes that the greatest payoff from reuse is realized in decreased maintenance costs. He reports maintenance cost reductions of up to 90 percent when reusable code, code templates, and application generators have been used to develop new systems.

While numerous organizations are achieving success in software reuse, not all organizations would be justified in making large investments in reuse due, for example, to the lack of commonality within their software projects (present, and anticipated). We will consider this point further in later sections.

1.5 SCOPE AND ORGANIZATION OF THIS DOCUMENT

In the following chapters we provide a discussion of many management and technical issues pertaining to software reuse, including various approaches that have been used or suggested for dealing with the issues. We distill the suggestions into guidelines for use. While some of the guidelines are quite specific (e.g., those for programming in Ada), most are not. Recognizing that reuse is still a very new, evolving field, we offer the guidelines as suggestions only. We hope that the discussion and guidelines will be helpful in understanding the issues involved and will serve as a starting point for undertaking software reuse. Each organization involved should eventually establish its own set of guidelines.

In the next two chapters we consider reuse within the broad categories of managerial and technical approaches. Chapter 2 focuses on managerial aspects of reuse, giving consideration to some existing impediments to reuse and approaches to their resolution, to the creation of positive incentives, and to the important issue of instituting a software development and maintenance process incorporating reuse.

Chapter 3 focuses on technical aspects of reuse and provides approaches and guidelines covering the activities of domain analysis, preparing reusable components (spanning the entire software life cycle), assuring component quality, and classifying and storing components. Specific guidelines are provided for developing Ada code modules for reuse. Approaches and guidelines for the reuse of software components are given, including searching, retrieving, understanding, assessing, adapting, and assembling components. Tools and environments for software reuse are also discussed.

Chapter 4, "Getting Started," contains some suggestions for a sequence of activities to implement reuse within an organization—from a modest beginning to a mature software reuse program. Carrying out the suggested activities of Chapter 4 requires dealing with the issues to which the guidelines of Chapters 2 and 3 apply.

Appendix A provides a complete list of the managerial and technical guidelines provided throughout this document. The technical guidelines are presented relative to the software life cycle.

2. MANAGERIAL GUIDELINES

In this chapter we consider some of the issues an organization's management must face to be successful in software reuse. Section 2.1 considers such issues as management approaches and organization; behavioral issues; and contractual, legal, and financial considerations. In Section 2.2 we discuss integrating reuse into the software development and maintenance process.

2.1 MANAGERIAL ISSUES AND APPROACHES

Many issues must be resolved to achieve a successful software reuse program. There are organizational, economic, legal, and sociological issues (Fairley et al. 1989), as well as issues of technology transfer, politics, tradition, and the continual advancement of technology (Aharonian 1989). Some technical reuse issues need to be resolved as well. The exact nature of issues naturally will vary from one organization to another, and thus means to resolve issues must be tailored to each organization. For example, DOD agencies have different legal issues to face than do their contractors. Thus we consider a broad spectrum of issues, with suggestions for eliminating (or alleviating) disincentives and for introducing incentives.

2.1.1 Leadership and Organizational Structure

Top-level management must take positive action to make software reuse a reality. This means much more than just issuing an edict that software reuse will occur. It means committing necessary resources to bring about a different way of approaching software development and maintenance—including a different process, tools, a well-trained support staff, and an adequate initial library of reusable components. It means spending seed money for later gains. Management must be prepared to wait for the investment to begin earning returns. Realistic goals must be set, and risks must be accepted.

Technical personnel must know that management is firmly committed to reuse, that perfunctory efforts are not acceptable, and that success in reuse will bring positive career rewards. Too many good new ideas die within organizations because management expects technical personnel to carry all the burden, including work over and above usual duties, with little or no support from management and sometimes with only negative incentives. One can guarantee that software reuse will never really flourish if the burden is on technical personnel alone. They simply cannot accommodate the necessary efforts to achieve success in reuse unless management provides resources and realistic assignments that take into account the expectation of reuse activities.

Top-level management must initiate planning and decision making for reuse, including decisions on scope for reuse (what organizational components will be involved, what life-cycle phases will be supported for reuse, and what application areas will be addressed). Organizational structure and behavior must be addressed. Contractual and financial approaches likely will be necessary. A decision with major implications is the process the organization will use for software development and maintenance (see Section 2.2). Training in software reuse must be provided to managers as well as technical personnel, and recruitment of a few strategic professionals may be advisable. Managers must be rewarded for saving money in their projects through reuse.

Managers in federal organizations are in an especially strong position to bring about reuse—not only in-house, but on the part of their contractors. There are

8 SOFTWARE REUSE GUIDELINES

currently substantial difficulties to overcome, as we will discuss in the following subsections, but markedly improved software productivity and quality are the prospective results of the undertaking. The best motivator for both management and technical personnel will be witnessing success. To that end it is very important that management carefully plan their reuse program before initiating it, including choice of early projects that seem likely to have a good payoff from reuse.

Fairley et al. (1989) have researched how to organize software reuse. They discuss functional, matrix, and project approaches, and conclude that the best approach is a "reuse matrix" structure, with domain-specific reuse groups. One or more individuals from domain reuse groups would be assigned to each project as a reuse "facilitator" for software relating to his/her domain of expertise. Each domain reuse group would be responsible for maintaining a library of reusable components, for educating and assisting in the use of available components for the domain, and for encouraging creation of additional reusable software. The following guidelines summarize some recommendations; most are from Fairley et al.

- LS1: *Upper-level management must set reuse goals, create an organizational infrastructure to support software reuse, establish policies, and provide necessary resources.*
- LS2: *Mid-level management must develop procedures, populate the organizational structure, allocate resources, and establish controls and metrics to achieve goals.*
- LS3: *Project-level management and technical personnel must carry out the reuse/reusability activities in individual projects.*
- LS4: *Establish an organizational entity whose charter is to promote reuse considerations at the corporate level.*
- LS5: *Evaluate the suitability of establishing the reuse matrix structure and domain reuse groups.*
- LS6: *Structure software development by domains amenable to reuse considerations.*
- LS7: *Establish strong connections between reuse and maintenance activities.*
- LS8: *Provide different types of training for managers, developers, and domain reuse specialists.*
- LS9: *Make personnel assignments that take reuse and reusability into account.*
- LS10: *Assign reuse facilitators to development groups.*
- LS11: *Allow two to three years after initiating software reuse before expecting economic advantages from the program.*
- LS12: *Provide a corporate financial "safety net" for projects practicing reuse; provide funding for generation of reusable components.*
- LS13: *Managers in federal organizations must take the initiative to influence the adoption of reuse within their own organizations and within contracts they direct.*

2.1.2 Organizational Behavior

A number of disincentives prevent personnel from cooperating in using available existing software and in preparing and supplying their own products for use by others. Fairley et al. (1989) mention some disincentives to using reusable components:

"not invented here" syndrome, more "fun" to build than adapt, differing styles and quality criteria, and technical constraints on the product. They mention the following as disincentives to contributing reusable components: schedule constraints, stylistic issues, reward structure, and peer pressure. They also strongly emphasize the importance of reuse being a corporate concern, with impetus for reuse coming from top-level management, i.e., a corporate culture must be established that emphasizes reuse.

To encourage individuals to participate effectively in reuse activities, one approach is to provide behavioral incentives, such as sharing cost savings, time off, bonuses, free dinners, reserved parking, public awards, and reuse bonus points. More effective, according to Fairley et al. (1989), is to seek to enhance psychological job satisfaction to motivate willing participation in reuse. Examples they suggest are: to provide corporate-level support for reuse as a meaningful endeavor; to reuse work products at higher levels of abstraction than code; to define carefully reuse roles for all involved personnel; to determine job performance through a metrics program; to provide for professional growth through job rotation and skills acquisition; to practice information hiding and object-oriented development to provide some autonomy; and to emphasize/train to give and accept constructive criticism. Fairley et al. make the important point that many of these approaches to enhance participation in reuse also promote good software engineering practices.

The above suggestions are briefly summarized in the following guidelines.

OB1: *Provide incentive rewards to participate in reuse.*

OB2: *Seek to enhance psychological job satisfaction to motivate willing participation in reuse.*

2.1.3 Contractual and Legal Considerations

Substantial disincentives regarding software reuse exist relative to contracting arrangements and ownership rights. Cost-Plus contracts at present provide disincentives to contractors to make use of reusable products. In fact, they provide incentives to redevelop rather than to reuse. Firm Fixed Price (FFP) contracts are worse regarding creation of reusable products—at least, for delivery to the customer—since it costs extra money to prepare products for reuse. There might be motivation to reuse available products (if such existed) in a FFP contract, as this could lead to greater profit. It could also be, of course, that effort could be expended under internal company funds to generalize some software developed under the contract for the company's later advantage.

The government must make it attractive to companies to create reusable software and to reuse software by providing incentives. A suggestion is to provide incentives through extra awards for contributions to, or applications of, reuse as value engineering similar to the hardware program (Joiner 1989). Companies could be required (or encouraged) to address applicable reusable software in submitted proposals. If such changes in government procurement policy are to be made, then accompanying changes must occur in how government projects are funded. At present no funds are allocated to a project to prepare software components for future use.

It appears that a company must always retain some proprietary software, for competitive advantage. To the extent software is placed in a repository for general use, a company must be duly compensated for the expected loss of the revenue that retention of the software would have given them (i.e., royalties must be paid). Hall (1987a) mentions the idea of a "meter" within a software component, that counts

10 SOFTWARE REUSE GUIDELINES

uses, and charges accordingly. Clearly DOD and other agencies must resolve the legal and contracting issues involved, for the national benefit.

Baker and Deeds (1989) argue that the government should not attempt to establish libraries of reusable software except in the cases in which a government organization does its own software development. They also raise the issue of the government's liability in the case of government-furnished software. They believe that government libraries will tend toward obsolescence, because the government has less incentive than industry to maintain state-of-the-art libraries. They further state:

Government should not tell corporations how to reuse software or make them use governmental libraries. If reuse makes sense, they will do it. They do not need us to make them do this. Double-billing for reused software should be ameliorated by increased competition and increased emphasis on establishing productivity baselines and measures.

There do seem to be situations, however, in which the government should build up libraries of contractor-developed software. One situation is a large project in which a number of contractors are involved and in which each contractor depends on a common subset of software. An example is strategic defense, in which many contractors are developing software, and without a government-held library, much duplication will likely occur.

There are several levels at which the government can promote reuse and reusability:

1. Encourage reuse internally in a government organization, where its library is integrated with its software process (Section 2.2).
2. In a contract with a single company, encourage reuse and reusability by procurement policy, but let the company decide how technically to handle library and process.
3. In a contract (or related contracts) spanning multiple companies, encourage sharing by procurement policy, as well as company-internal reuse; in this case it may be necessary for the government to establish a government-managed reuse library to serve the group of companies.

One issue relating to libraries serving multiple companies is that if software components other than code are reused, they may be tied to a specific software process of the company that created the components, and thus might be unwieldy for use by other companies. For example, designs based on functional decomposition and designs based on the object-oriented approach could be awkward to integrate. Here again the government may influence the process to some extent by procurement policy.

It is clear that some changes are necessary to the government procurement process in order to encourage reuse and reusability. The STARS *Reusability Guidebook* (Wald 1986) appears to be the most complete coverage of the government procurement process relative to reuse prepared to date. Many legal issues are as yet unresolved pertaining to software reuse, particularly those concerning ownership rights in software (for example, when developed under contract to the government) and liabilities for errors when software is reused by other organizations. These issues may be resolved by legislation, by stipulations of individual contracts, or by the results of litigation over time. More research is needed to formulate specific recommendations for resolving this important issue.

The above observations lead us to the following guidelines.

- CL1: *Seek contractual means to require or to encourage contractors to create reusable software (from which the government gains future benefits) and to reuse existing software (which gives the government immediate benefits).*
- CL2: *Establish and enforce reuse practices within government development groups.*
- CL3: *Require reuse within a group of related contracts (e.g., by a prime contractor and subcontractors).*
- CL4: *Seek means to alter project funding approaches to encourage creation of reusable software.*
- CL5: *Seek resolution to the legal issues of potential liability and partial ownership.*

2.1.4 Financial Considerations

Because of the extra effort required to generalize and to test the components, it costs more to prepare software for reuse than for a single use. One could easily spend more money trying to reuse ill-suited components than in preparing custom software. In order to assess opportunities for reuse and reusability, cost predictions must be made. Barnes et al. (1987b) present a framework for analyzing reuse cost. Underlying the costing must be a database of financial data on previous projects. Unfortunately most organizations have no such record of project performance data, but one can be built over time.

While there are several cost models in use for the general software process, little work has been done in cost modeling for reuse/reusability assessment. As part of the AIRMICS Ada Reuse and Metrics project, George Mason University and CONTEL Technology Center surveyed available software cost models relative to reuse, and prepared an extensive bibliography (see Fairley et al. 1989). They concluded that Ada-COCOMO is the best available cost model for modeling Ada development—the model incorporates costs for developing reusable components, and savings from reusing components. The model also incorporates incremental development as an option. A major shortcoming of Ada-COCOMO is that it models reuse on a project-by-project basis. In fact, the researchers could find no existing cost model that treats reuse as an organizational issue rather than a project-level issue.

CONTEL researchers Pfleeger and Bollinger (Fairley et al. 1989) devised an economic model and an associated cost model based on the concepts of baseline projects, producer/consumer reuse considerations, broad spectrum reuse, and reuse cost amortization. They suggest a design for a reuse cost estimation tool. They developed terminology and a viewpoint to support consideration of reuse issues at the corporate level. The interested reader is urged to obtain the CONTEL portion of Fairley et al. 1989, to understand fully the ideas developed.

Fairley et al. (1989) recommend the use of Ada-COCOMO in the short term, with development of economic models/cost models at the corporate level, in the longer term.

We offer the following guidelines on reuse financial considerations. Several are from Fairley et al. (1989).

- F1: *Establish mechanisms to accumulate an organizational database of historical financial data relative to software production and maintenance, including reuse activities.*
- F2: *Provide cost modeling tools, to the extent feasible, in concert with organizational data for reuse/reusability decision assessments (including make versus reuse versus buy decisions).*

12 SOFTWARE REUSE GUIDELINES

F3: *Consider/model costs over multiple projects.*

F4: *Establish mechanisms to share the cost of developing reusable components across multiple projects.*

2.2 SOFTWARE DEVELOPMENT AND MAINTENANCE INCORPORATING REUSE

2.2.1 Software Process Concepts and Issues

Software reuse cannot occur except as part of an integral framework for software development and maintenance. DOD-STD-2167A guides development of software for the government. Taylor (1989) has correctly observed, "Although the practices detailed in this standard do not preclude extensive use of software components, the life-cycle model and data requirements provide little support for or active encouragement of software component technology." The problem is no doubt more fundamental than the DOD standard, per se, in that if most government and industrial organizations practice reuse at all, it is in an ad hoc, nonsystematic way. Each organization needs a systematic approach to the development and maintenance of software that includes reuse and reusability as important, integral, natural, and inescapable elements.

In order to be sure of the use of terminology, consider the following definitions from Humphrey (1989):

- * Software refers to a program and all of the associated information and materials needed to support its installation, operation, repair, and enhancement.
- * Software engineering refers to the disciplined application of engineering, scientific, and mathematical principles and methods to the economical production of quality software.
- * The software engineering process is the total set of software engineering activities needed to transform a user's requirements into software.

Humphrey comments that this process may include, as appropriate, requirements specification, design, implementation, verification, installation, operational support, and documentation, as well as temporary or long term repair and/or enhancement (i.e., maintenance) to meet continuing needs. There is a great deal of research relative to the software process, as evidenced by the ongoing Software Process Workshops (e.g., Tully 1989).

The waterfall model, inherent in DOD-STD-2167A, is a top-down, single-project view of system development. Simos (1987) has noted:

- * Software reuse is not inherently "top down."
- * Reuse involves a perspective that necessarily looks beyond the development of single projects or systems.
- * Reuse involves the exploitation of commonality at many levels of abstraction besides that easily captured in code.

Simos (1987) further observes:

Software reuse is not specifically addressed in the conventional top-down model, in which systems are designed via a process of modular decomposition. Hence, structured methodologies rarely include specific techniques for analyzing domains to extract maximally reusable components (though

object-oriented design methods may rightly claim to improve this state of affairs to a great extent (Simos references Meyer 1987 here)). Since most applications are in fact hybrids of reuse and new design, *de facto* reuse does occur; yet it is often confined to ad hoc, informal contexts that fall between the cracks of the "official" methodology being followed by the project, and is dependent on developers' intuitive grasp of common functionality within the semantics of the application.

What is needed is a process model that allows for iteration between the top-down, "problem-driven" approach and a bottom-up, "parts-driven" perspective ... Such a process model would correspond more closely to the real state of practice in software development than the current model, and would at least initially have a less prescriptive, more descriptive, flavor.

2.2.2 A Generic Reuse/Reusability Model

Kang (1987) suggests a refinement to the DOD-STD-2167A life cycle by identifying reuse activities applicable to each phase. He describes a generic reuse activity model developed at SEI as the base model for use in refining each phase. As noted in Chapter 1, Kang defines reuse to be "an act of synthesizing a solution to a problem based on predefined solutions to subproblems." He proposes the following four steps to be performed at each phase:

1. understanding the problem and identifying a solution structure based on the predefined components;
2. reconfiguring the solution structure to improve the possibility of using predefined components available at the next phase;
3. acquiring, [assessing,] instantiating, and modifying predefined components; and
4. integrating the components into the products for this phase.

Note that we have added "assessing" to Kang's statement number 3. While he mentions "assessing" in his discussion and his statement therefore assumes assessment is performed, we felt "assessing" should be explicitly stated for clarity. Kang offers the following comments on the four steps:

The major tasks under the first step are to understand the problem to solve, build up the knowledge of the predefined solutions, and apply the knowledge in structuring the problem in terms of the subproblems to which solutions already exist.

Once a solution structure is identified based on the predefined components available at a given phase, the next step is to reconfigure the solution in order to optimize reuse both at the current phase and the next phase. Doing so requires identifying experts of the next phase activity who will review the proposed solution, identify candidate components available at the next phase, and evaluate the reusability of the candidate components. Based on the potential reuse at the next phase as well as at the current phase, an optimal solution structure is to be identified. We anticipate that the first two steps would be iterated a number of times.

The major output from the first two steps are a solution structure and a reuse plan for the next phase. The third step includes tasks of making

14 SOFTWARE REUSE GUIDELINES

components identified in the solution structure ready for integration. These tasks include acquiring reusable components, modifying and/or instantiating reusable components, and developing the components that cannot be acquired or for which modification is not economic. Finally, the completed components are integrated into the product(s) required for the phase. The products are subjected to a formal review before being released to the next phase.

Kang (1987) also includes an example of the application of this generic reuse model to the software requirements phase. That discussion identifies the experts who will participate in step 2 as domain experts. The reuse model would be applied to refine the other phases of the 2167A life cycle in much the same way. SEI is currently evaluating this approach to integrating reuse into the software process.

As the generic reuse activity model now stands, it does not deal with the contribution of reusable components to a library. The following is suggested as step 5 (thus creating a five-step reuse/reusability model):

5. evaluating reusability prospects of components that must be developed and components obtained by modifying predefined components for contribution to the set of predefined components.

This activity would include consideration of the advisability of generalizing the components for improved reusability.

It is significant to observe that this reuse/reusability model actually spans multiple projects, since the expectation is that products created within a given project will be used in other projects. Thus to be effective in software reuse, an organization must of necessity take this multi-project view of development and maintenance.

Decisions with respect to reuse must be based on knowledge, and automation of the decision process is an important goal (e.g., Iscoe 1987). The process proposed by Kang (1987) assumes a manual decision process, based on the use of domain experts. Research is being conducted to determine better ways to perform domain analysis and to capture domain knowledge in machine-processable form. (We discuss domain analysis in Section 3.1.) Kang's approach does not explicitly take into account the transformational approach to the life cycle. The goal of generating executable software from requirements has been realized so far only in very limited domains (e.g., see the discussion of MicroSTEP in Section 2.2.3). To the extent success is achieved, it is unnecessary to go through the life cycle phases manually.

Dusink (1989) takes a somewhat different approach than Kang. He suggests a set of steps that go through the entire software development process, focusing on the application domain. Dusink also discusses supporting tools for his process.

Basili et al. (1989) characterize the reuse process as:

1. specifying the object to be created;
2. searching the project, domain, and general databases for reuse candidates;
3. evaluating the candidates to determine which (if any) should be used;
4. modifying, if necessary, to fit specific needs;
5. integrating the reusable components;
6. validating; and
7. feeding back the knowledge regarding the payoff of reuse.

2.2.3 Alternatives to the Waterfall Model

Perhaps the best-known alternative to the waterfall life-cycle model is the spiral model (Boehm 1988), which recognizes that software development is not inherently sequential. The model incorporates risk analysis at each spiral phase. Boehm has not addressed reuse explicitly with the spiral model.

Simos (1987) proposes the domain-oriented software life cycle. The domain life-cycle model "formalizes typical patterns in the development of related series of applications and the persistence of information from one application to the next" (Simos 1987).

Raymond Yeh and his colleagues are actively pursuing software reuse in connection with the prototyping life-cycle paradigm (Yeh and Welch 1987). Hartman (1989) describes their approach, which is based on incremental interactive specification with prototyping performed at each level of development, with reuse at each level (specifications, design, code, etc.), and with maintenance performed at the specification level. Their developments include PROTO (a tool for functional prototyping) and MicroSTEP (which achieves mapping from high-level specifications to code for a specific domain). They are also engaged in U.S. Army Strategic Defense Command (SDC)-funded work to define and to implement methodology, environment, and tools to support the life-cycle model.

An increasingly important methodology is OOD. It is considered by many researchers to be promising relative to software reuse. Booch (1987) combines OOD with component reuse through use of the Ada programming language, and has spurred a great deal of interest in the promise of reuse. Another influential advocate of OOD as the basis for reuse is Meyer (1987). His Eiffel language serves as the basis for his reuse research and recommendations. McKay (1989) has proposed an object-oriented approach to software development and maintenance, with an automated, knowledge-based approach to identification of reuse opportunities.

Lieberherr and Riel (1988) have designed the Demeter system based on OOD, coupled with parameterized classes. They seek to "grow" software (as recommended by Brooks (1987)) through inheritance and parameterization, rather than to build or to create software.

Kaiser and Garlan (1987) have sought to improve OOD for reuse by devising a notation (called MELD) that is independent of any object-oriented language (and would be translated into a conventional programming language). Their system supports composition of components through merging of data structures and methods from two or more "features" (their name for reusable building blocks similar in concept to Ada packages). They employ inheritance and data structure/behavior encapsulation from OOD.

Rogerson and Bailin (1987) conducted an experiment in reuse based on OOD versus functional decomposition and determined that it is easier to detect reusability within a given context for objects (which they represented as Ada packages).

Bailin (Bailin 1987; Bailin and Moore 1987) describes an object-oriented approach to software development incorporating reuse, devised by Computer Technology Associates, Inc. for NASA GSFC. They anticipated that OOD could foster reuse, and recommended that GSFC pursue the approach. They have developed the Software Reuse Environment (SRE), consisting of an OOD facility, a reuse database, and search and navigate functions. The OOD facility is the user front-end, and provides graphics support for diagram representation, automatically creates a first approximation to an OOD for annotated data flow diagrams, and assesses the quality of hierarchies of object diagrams (by means of the design quality assessor). The reuse database is structured by means of semantic networks (relating

16 SOFTWARE REUSE GUIDELINES

objects by keywords and associations, constituting a "faceted classification scheme" (Prieto-Diaz and Freeman 1987)). This concept is discussed further in Chapter 3. Navigation functions permit a user to begin with a specified product or keyword and to move through the reuse database examining relationships and attributes of reusable objects for possible reuse.

Bailin is also performing research toward applying an object-oriented approach to requirements specification (Bailin 1989) with the intention that the object-oriented specification lead smoothly to OOD and thus to programming in Ada or some other high-level language. He suggests using this method rather than structured analysis when the use of OOD is anticipated. Domain analysis may be viewed as a life-cycle phase occurring prior to requirements determination (Kang 1989; Prieto-Diaz 1987a). Domain analysis is discussed in Section 3.1. Domain analysis would not precede each requirements determination activity, but rather its output (i.e., domain model and domain architecture) would serve as input to the requirements determination activity for numerous software development projects.

2.2.4 Establishing a Process

Substantial progress is being made in devising software engineering processes that encourage reuse and reusability. While specific aspects of some approaches are presently undergoing experimental evaluation, there can be no doubt that many of the ideas are workable and can be applied with profit. Kang's generic reuse activity model, augmented with a reusability step as suggested herein, seems a very practical and workable approach to 2167A refinement. And, it would appear to be applicable as stated to any multiphase process (e.g., Bailin's object-oriented process) in which phase transitions are based on manual decision making. Indeed, the same process would be required in the event of automated support to phase transitioning—the only difference being that some of the activities would be achieved by automated, knowledge-based means.

Maintenance activities were not directly addressed by Kang, but the reuse/reusability model applies equally well to software enhancement as to original development, since maintenance and original development should follow essentially the same software engineering process.

As we have previously noted, deciding on the software process (or processes) for an organization is a major decision. Consideration of Kang's model makes it clear that reuse likely can be accommodated effectively as a refinement to an organization's existing software process, perhaps lessening to some extent the impact of introducing software reuse. Whatever approach to a software engineering process is taken, software reuse and reusability should be important, integral, natural, and inescapable elements of the process. The following guidelines ensue from the preceding discussion.

- SP1:** *Initiate action to establish a software engineering process (including development and maintenance) that includes reuse and reusability as important, integral, natural, and inescapable elements.*
- SP2:** *Augment DOD-STD-2167A with refinements that specifically support and encourage reusability and reuse (e.g., the five-step generic reuse/reusability model).*
- SP3:** *Consider the object-oriented methodology for use within the software process.*
- SP4:** *Automate activities within the software process as understanding and experience permit.*

3. TECHNICAL GUIDELINES

3.1 DOMAIN ANALYSIS

A few organizations have taken all the software being developed and placed it into a repository (which is sometimes appropriately called a software junkyard). The software components considered for reuse, however, should be carefully chosen, considering that often the reuse development costs can be considerably higher than the development costs for the custom software. This may be due to the form the reusable software takes (generic, parameterized, application generator, etc.) and to the rigor employed in testing it.

Two categories of software seem to be good candidates for reuse: horizontally reusable and vertically reusable components. Horizontal reuse refers to reuse across a broad range of application areas (such as data structures, sorting algorithms, and user interface mechanisms), while vertical reuse refers to components of software within a given application area that can be reused in similar applications within the same problem domain (Tracz 1987a). Horizontal reuse has, no doubt, been studied the most so far (Booch 1987), and it likely has occurred much more frequently than vertical reuse. The main reasons for this are that horizontal reuse is better understood and easier to achieve. On the other hand, the greatest potential leverage can come from vertical reuse—by intensive reuse of carefully crafted solutions to problems within an application domain. The CAMP project (McNicholl et al. 1986) is an example of vertical reuse.

In order to achieve vertical reuse, a domain analysis is required. Kang (1989) reports on domain analysis work (sometimes called features analysis) at SEI. He describes domain analysis as follows:

Domain analysis is a phase in the software life cycle where a domain model, which describes the common functions, data and relationships of a family of systems in the domain, a dictionary, which defines the terminologies used in the domain, and a software architecture, which describes the packaging, control, and interfaces, are produced. The information necessary to produce a domain model, a dictionary, and an architecture is gathered, organized, and represented during the domain analysis.

Domain analysis is related to requirements analysis but it is performed in a much broader scope and generates different results. It encompasses a family of systems in a domain, produces a domain model with parameterization to accommodate the differences, and defines a standard architecture based on which software components can be developed and integrated. A domain model and an associated dictionary represent the domain knowledge, and an architecture represents the framework for developing reusable components and for synthesizing systems from the reusable components. An ideal domain model and architecture would be applicable throughout the life-cycle from requirements analysis through maintenance.

Kang (1989) also answers the question, "Why do we need to do domain analysis?"

As the areas to which computers are applied become larger, one of the problems faced by the industry is that it is often difficult to find software engineers who have the required application domain knowledge. Reuse of

18 SOFTWARE REUSE GUIDELINES

application domain knowledge is becoming an important issue in software engineering. The purpose of domain analysis is to gather and represent application domain knowledge in a model and to develop an architecture that shows how the problems in a domain are addressed in software systems. A domain model unifies and consolidates the domain knowledge which may be reused in subsequent developments.

More and more organizations consider software as an asset that can provide an important edge in business competition. Therefore, identifying areas that will maximize the return on software investment is an activity that encompasses both business planning and software engineering. The business planning activity identifies future products, and the domain analysis activity identifies the product commonality and potential software assets. The information on the software assets can be fed back to future business planning. Also, the product commonality information enables large-grain reuse across the products.

The productivity and quality improvement from reusing components built for the purpose of reuse is much greater than that from components developed without reuse in mind. However, in order to build reusable components, the contexts in which the reusable components will be used must be understood and the reusable components must be designed to accommodate the contextual differences. A domain model and an architecture define the contexts for developing reusable components.

Kang (1989) also briefly summarizes an approach to domain analysis and provides the following summary of the paper:

Domain analysis is an activity to produce a domain model, a dictionary of terminologies used in a domain, and a software architecture for a family of systems. These outputs from the domain analysis:

- * facilitate reuse of domain knowledge in systems development,
- * define the context in which reusable components can be developed and the reusability of candidate components can be ascertained,
- * provide a model for classifying, storing, and retrieving software components,
- * provide a framework for tooling and systems synthesis from the reusable components,
- * allow large-grain reuse across products, and
- * can be used to identify software assets.

Based on our experience with a domain analysis (called features analysis in this project) and the potential benefits from it, we believe that domain analysis should be a standard activity in the software development life cycle.

Lee and Rissman (1989) describe their work at SEI in determining domain-specific software architectures.

Cohen (1989) summarizes a domain analysis methodology in eight steps. He also shows the steps in flow diagram form. The steps are:

1. Select specific functions/objects.
2. Abstract functions/objects.

3. Define taxonomy.
4. Identify common features.
5. Identify specific relationships.
6. Abstract the relationships.
7. Derive a functional model.
8. Define a domain language.

A domain language could take one of many possible forms: a conventional language, a knowledge-based system, an object-oriented system, or another available formalism.

Prieto-Diaz (1987a) comments on the need to define a domain boundary—i.e., where one domain ends and another begins. Also, he views the domain-specific language, produced in step 8 above, as encapsulated in a formal language and serving as a specification language for the construction of systems in the domain. Prieto-Diaz characterizes this as the “reuse of analysis of information,” and states the opinion that this “is the most powerful sort of reuse.”

Prieto-Diaz briefly summarizes the domain analysis approaches used by Raytheon (in the work reported by Lanergan and Grasso 1984), and by McDonnell Douglas (in the CAMP work). Palmer (1989) of McDonnell Douglas provides the following observations from the CAMP work:

CAMP-1 began with a domain analysis that involved the missile operational flight software from a set of ten missiles. From this analysis, we identified approximately 250 common parts and developed a taxonomy with which to categorize those parts. We assumed (and were proved correct) that we would identify additional parts once we actually began development of the common parts. Our final part count at the end of CAMP-2 was 454.

There are as yet no widely accepted or established techniques for performing a domain analysis, but a number of issues have been identified. One factor that is critical is the selection of an adequate domain representation set upon which to base the analysis. Practical constraints prevent the examination of all applications within a domain, thus, it is important that the sample set include applications that are truly representative of the domain as it has been defined.

Neighbors (1987) reports on the deliberations of the Domain Analysis Working Group at the Workshop on Software Reuse, held in October 1987. The report states that “given a domain analysis, an organization should be able to: (1) use the domain model to check the specifications and requirements for a new required system in the domain; (2) educate new people in the organization providing them with the general structure and operation of systems in the domain; and (3) derive operational systems directly from the statement of the system in domain specific terms.”

The working group undertook the domain analysis of library management systems as a practical problem, with the same individuals serving as both domain experts and domain analysts. Neighbors (1987) describes the group's activities in some detail and concludes by giving the following “Basic Domain Analysis Process.”

1. Establish the domain subject area.

20 SOFTWARE REUSE GUIDELINES

2. Collect the domain experts.
3. Establish the depth of analysis (i.e., whether to analyze subdomains).
4. Establish the width of analysis (i.e., determine the boundary of the domain—"Is this function required by most of the systems built in this domain?").
5. Define the specific domain objects, operations, relationships, and constraints.
6. Hand test the domain by attempting a description of a specific system in the domain.
7. Package the domain for constructive reusability by expressing it in a form for a transformational refinement tool such as Draco.

The working group members used various analysis representations, including data flow diagrams, entity-relationship diagrams, semantic nets, object diagrams, and class hierarchies with inheritance. They concluded that usually only one each from the object hierarchy, data flow, and control flow representations would be needed.

Hutchinson and Hindley (1988) report on their work in developing a domain analysis method. Their goals were:

- * to discover the functions that underwrite reusability,
- * to focus the domain specialist's attention on reuse,
- * to help the domain specialist ascertain reuse parameters,
- * to discover how to redesign existing components for reuse, and
- * to organize any domain for reuse.

The domain analysis was done by a reuse analyst with the assistance of a domain specialist—an individual with an excellent understanding of the problem domain. The researchers developed structured domain analysis techniques based on questions devised to assess a software component's reusability. The domain on which they based their experimentation was a simulation of the utility systems management (USM) system of the Experimental Aircraft Programme (EAP) in the United Kingdom. The subdomains they considered were propulsion, fuel management, and undercarriage. In the case of propulsion, this subdomain was considered for reuse because the controlled hardware (the engines) would not change significantly between the EAP implementation and the next project; fuel management was chosen because the domain appeared to contain a lot of functional duplication within the requirements definition. Undercarriage was chosen because much of its operation would not change on future implementations.

The reuse analyst decided on three levels of reuse to clarify the domain: the initial level pertained to reuse of the whole system, the next level to reuse of subsystems, and the final level to functions at the requirements level and to components at the design and code levels. The reuse analyst presented 12 questions to the domain specialist, based on the assumption that it is domain-specific knowledge that can isolate reusable components. The questions seek to elicit identification of reuse attributes and reusable components in an understandable manner. The questions are:

- * Is component functionality required on future implementations?

- * How common is the component's function within the domain?
- * Is there duplication of the component's function within the domain?
- * Is the component hardware dependent?
- * Does the hardware remain unchanged between implementations?
- * Can the hardware specifics be removed to another component?
- * Is the design optimized enough for the next implementation?
- * Can we parameterize a nonreusable component so that it becomes reusable?
- * Is the component reusable in many implementations with only minor changes?
- * Is reuse through modification feasible?
- * Can a nonreusable component be decomposed to yield reusable components?
- * How valid is component decomposition for reuse?

The authors observe that reuse proved to be practical, even in the hardware-dependent areas being analyzed. They assessed the requirements functions as potentially 75 percent reusable for the next implementation, and indicated that reuse could be equally high for code designed for reuse from these requirements.

Tracz (1987a) also goes through an example of domain analysis, and the subsequent reusable software design, based on his use of both parameterization and application generators. His general approach to software composition is summarized in Section 3.3.4.

We conclude this section by suggesting the following guidelines.

- DA1: *Select domain(s) carefully for analysis, based on the maturity and stability of the organization's activities within each domain and on the planned emphasis the domain is to receive.*
- DA2: *Determine and apply a systematic approach to domain analysis, yielding a domain model, a set of domain terminology, and a domain architecture.*
- DA3: *Use domain analysis results as a basis for classifying, storing, and retrieving reusable components.*
- DA4: *Use domain analysis results as a basis for decisions about the advisability of investing in specific instances of reusable software.*
- DA5: *Use domain analysis results to help understand how existing domain-specific reusable software may be applied.*

3.2 CREATING REUSABLE COMPONENTS

3.2.1 Reusable Components Spanning the Life Cycle

The goal of practitioners of software reuse should be to make use of existing knowledge and software artifacts throughout the software life cycle (wide-spectrum reuse), with the expectation that, the earlier the reuse occurs in the life cycle, the greater should be the payoff. Code reuse is better understood and more prevalent by far than other levels of reuse, but reuse of other kinds of software components is accelerating.

22 SOFTWARE REUSE GUIDELINES

As we have previously noted, it costs extra to prepare software components for reuse due to the necessary effort to generalize the components, to conduct extra testing, to document the components, and to classify and to store them for reuse. Thus it is important that a careful assessment be made of the likely payoff of such extra costs, before investing in this component. This decision process is inherent in step 5 of the suggested reuse/reusability model in Section 2.2.2.

Reuse of domain knowledge is a very high level of reuse, and can give extremely important leverage in productivity and quality, as we discussed. As we noted in Section 3.1, a "domain language" is a very high level language incorporating knowledge of the objects, operations and relationships within a domain, which can serve as a specification language for use in constructing systems in the domain (Prieto-Diaz 1987a). This is reuse of "meta-requirements," in a sense. The software architectures for a given domain, as discussed in Section 3.1, also embody much in the way of requirements and design, and use of the architecture results in reuse of requirements and design. Domain analysis information is an important ingredient in the decision process concerning preparation of reusable components. That is, the required functionality within a domain, as reflected by the domain analysis, helps determine whether a candidate component's functionality is likely to be needed for reuse.

Lubars (1987) of MCC discusses the potential reuse of many kinds of software components, including domain-related knowledge, requirements, abstract designs, abstract algorithms, design and program transformations, and code. Also important for reuse are test plans, test cases, and test results. Research is being conducted to generate code from requirements through automatic transformations (e.g., Baxter 1987). Some success has been achieved in very narrow domains (e.g., Hartman 1989), yet in the foreseeable future manual transformations will be necessary in moving from phase to phase in the software life cycle. The greatest occurrence of high-level reuse presently is probably in "personnel reuse"—individuals who previously have analyzed requirements for similar systems, put to use their retained insights and approaches.

A great deal of "implicit" reuse takes place when products are reused. For example, when a code module is reused, a design is being reused, and to some extent requirements are being reused. Thus benefit is derived by not having to repeat preceding activities. In Section 3.2.2 we consider mechanisms for reusable code. There we will see that so-called "generators" actually incorporate (and thus reuse) a very great deal of requirements and design information.

A code component retained "in isolation" in a reuse library is likely to be of little value. Thus it is extremely important that design and requirements associated with code modules be retained, along with test plans, test cases, test results, prototypes (perhaps), and other related life cycle products. The associated set of components can be useful in many ways. For example, determining a match (or near-match) with reusable requirements, may lead to reuse of the associated design and code, or at least to reuse of the high-level design. Clearly, the high-level design should be easier to understand and adapt than the associated code would be. And, the high-level and detailed designs should be very valuable in understanding code.

Due to the likelihood that in the near term reusable components will be reused manually within the software process, the organization of the library, and the means for searching and retrieving related sets of candidate components, and understanding and adapting components, are very important considerations. In Section 3.2.5 we consider issues of classifying and storing reusable components; the issues of locating and adapting components are discussed in Section 3.3.

We end this subsection with the following guidelines.

- CC1: *Provide domain analysis results within the reuse framework—explicitly and/or implicitly.*
- CC2: *Make careful assessments, including financial predictions, in deciding whether to develop a reusable component.*
- CC3: *Prepare for reuse all more-abstract life cycle representations of a reusable component (e.g., prepare requirements specification for a high-level design).*
- CC4: *Record and supply adaptation suggestions with a reusable component.*
- CC5: *Generalize a reusable component to the extent practical during its preparation.*
- CC6: *State as a requirement the reuse of software and/or the creation of reusable software.*
- CC7: *During the requirements determination phase, conduct the following activities:*
 - 1. *Make use of available requirements components as appropriate.*
 - 2. *Structure requirements to take advantage of available high-level designs.*
 - 3. *Emphasize preparation of reusable requirements components (from newly developed requirements and modified, previously available requirements).*
- CC8: *During the high-level design phase, conduct the following activities:*
 - 1. *Make use of available high-level-design components as appropriate.*
 - 2. *Structure the high-level design to take advantage of available detailed designs.*
 - 3. *Emphasize preparation of reusable high-level-design components (from newly developed high-level designs and modified, previously available high-level designs).*
- CC9: *During the detailed-design phase, conduct the following activities:*
 - 1. *Make use of available detailed-design components as appropriate.*
 - 2. *Structure the detailed design to take advantage of available code modules.*
 - 3. *Emphasize preparation of reusable detailed-design components (from newly developed detailed designs and modified, previously available detailed designs).*
- CC10: *During the coding and unit-testing phase, conduct the following activities:*
 - 1. *Make use of available code components and test cases as appropriate.*
 - 2. *Emphasize preparation of reusable code components and test cases (from newly developed code/test cases and modified, previously available code/test cases).*
- CC11: *During integration testing, conduct the following activities:*
 - 1. *Make use of available test plans, test cases, and test results as appropriate.*

24 SOFTWARE REUSE GUIDELINES

2. *Emphasize preparation of reusable test plans/cases/results (from newly developed tests and modified, previously available tests).*

CC12: *Recognizing that the maintenance phase contains as subphases the software development phases, apply to maintenance the guidelines provided above for requirements, design, coding, and testing.*

3.2.2 Code Components

Reusable code is the most researched of the reuse abstraction levels; more is known about it, more has been written about it, and most software reuse has been based on code. In this section we consider two aspects of reusable code, namely (1) passive versus dynamic code components and (2) programming style.

3.2.2.1 Passive Versus Dynamic Code Components

There are two different clearly distinguishable categories of code components for reuse. The first may be called passive components or building blocks, which are used essentially unchanged, by means of composition. The second category involves dynamic components or generators, which generate a product for reuse. These components are also spoken of as reusable patterns, and they are very effective when feasible. Generators are of two fundamental types (Biggerstaff and Richter 1987): (1) application generators (employing reusable patterns of code) and (2) transformation systems (which generate a product by successive application of transformation rules; e.g., see Cheatham 1984). Generators by their nature tend to lift the abstraction level for reuse above that of code building blocks. For example, compiler syntax analyzer generators (e.g., YACC) do their work with little need for the user to understand the underlying concepts. Simulation languages/systems constitute another application of reusable patterns to achieve effective leverage (Hooper 1988). Prototyping usually is based on significant reuse of software; it may be very high-level language based, somewhat like simulation languages, and/or may be based on code blocks. Fourth generation languages constitute another kind of reusable pattern, and also substantially lift the reuse abstraction level.

Code blocks for reuse have historically been subprograms (procedures or functions), and now also include Ada packages (collections of reusable subprograms with their encapsulated environment), classes (in the sense of object-oriented programming), and Ada generics. Parameterized code may be based on any of the forms of code blocks just mentioned. Also, code templates may be used, with slots to be filled in by a user to customize for a given application.

The recent past has produced better programming languages for reusability support—especially Ada. Modula-2 and various other available languages also have some good features. But the DOD mandate for use of Ada doubtless makes Ada the language of choice for code reuse, and Ada has very strong features for reuse, such as the package (information hiding, encapsulation, etc.) and generics. (See Section 3.2.3.)

3.2.2.2 Programming Style

Basili et al. (1989) have provided a number of guidelines for developing Ada code, based on their research into the impacts for reuse of the strength of data bindings of a module with its environment, and on their assessment of effort necessary to transform existing Ada code for reuse in contexts other than the original one. They suggest the following:

- * Avoid excessive multiple nesting in any language constructs.
- * Usage of the USE clause is to be avoided if possible.
- * Components should not interact with their outer environment.
- * Appropriate use of packaging could greatly accommodate reusability.
- * Avoid mingling resource units with application-specific context.
- * Avoid literal values.
- * Keep interfaces abstract.

The guidelines above that are Ada-specific are also included in Section 3.2.3, Guidelines for Writing Reusable Ada Code.

Berard (EVB 1987) stresses the importance of good coding style for reuse—e.g., using meaningful identifiers; avoiding literal constants; using adequate, concise, and precise comments; making frequent and appropriate use of packages; isolating and clearly identifying environmentally dependent code; and, modules should be highly cohesive and loosely coupled. In general, as implied above, the approaches that promote understanding for maintainability serve equally well for reuse. Barsotti and Wilkinson (1987) argue that reusability of code is essentially a byproduct of quality and maintainability. To this end, they recommend:

- * using certified algorithms,
- * defining error handling,
- * defining exceptional conditions,
- * establishing explicit interfaces,
- * developing modular programs,
- * parameterizing, and
- * providing all test data and reports.

They also provide a list of quality criteria: accuracy, application independence, augmentability, completeness, conciseness, consistency, fault tolerance, generality, legibility, self-descriptiveness, simplicity, structuredness, system accessibility, and traceability. They provide precise definitions to all these terms. By application independence they mean nondependency on the database system, microcode, computer architecture, and algorithms (i.e., not independence from an application domain); this thus includes the important criterion of portability. By system accessibility, they mean provision for control and audit of access to the software and data.

They indicate that all these criteria support reusability and that all support maintainability except accuracy, application independence, and fault tolerance: reliability is supported by all except application independence, augmentability, generality, self-descriptiveness, simplicity, and system accessibility. While one could argue with some of these particulars, the concepts are worthwhile. These and other criteria relate to the concept of software metrics—criteria by which the suitability of software may be measured.

St. Dennis (1987) suggests a list of 15 language-independent characteristics of reusable code.

1. Interface is both syntactically and semantically clear.
2. Interface is written at the appropriate (abstract) level.

3. Component does not interfere with its environment.
4. Component is designed as object-oriented; that is, packaged as typed data with procedures and functions that act on the data.
5. Actions based on function results are made at the next level up.
6. Component incorporates scaffolding for use during building phase.
7. Information needed to use software, its specification, should be separate from the details of its implementation, its body.
8. Component exhibits high cohesion/low coupling.
9. Component and interface are written to be readable by persons other than the author.
10. Component is written with the right balance between generality and specificity.
11. Component is accompanied by sufficient documentation to make it findable.
12. Component can be used without change or with only minor modifications.
13. Component should be insulated from host/target dependencies and assumptions about its environment; it should be isolated from format and content of information passed through it that it does not use.
14. Component is standardized in the areas of invoking, controlling, and terminating its function; error-handling; and communication and structure.
15. Components should be written to exploit domain of applicability; components should constitute the right abstraction and modularity for the application.

We will consider these characteristics further with regard to the Ada reuse guidelines in Section 3.2.3. Now we suggest the following guidelines.

- CC13:** *Supply reusable code in the form of a generator or a transformation system for greater reuse leverage, when practical.*
- CC14:** *In preparing code blocks, use Ada generics, parameterized procedures, and code templates for greater reuse generality, as appropriate.*
- CC15:** *Emphasize good programming style in developing reusable code, creating code exhibiting understandability, reliability, and maintainability.*

3.2.3 Guidelines for Writing Reusable Ada Code

The Ada language is at once a primary motivator of reuse and a primary mechanism for reuse. Ada provides a rich set of features supporting reusability (St. Dennis 1986), including:

- * packages,
- * separate compilation and checking across program units,
- * separate specifications and bodies for program units,
- * information hiding constructs (e.g., private types),
- * generics, and

- * strong typing.

Ada was designed to promote reliability and maintainability, programming as a human activity, and efficiency. As St. Dennis (1986) notes,

Software reliability, maintainability and efficiency also contribute positively to reusability. Reliability contributes to user confidence in a software component, maintainability to understandability, and efficiency to feasibility for reuse.

As we have noted, a number of reuse guidebooks are available (e.g., Wald 1986; ISEC 1985; Braun, Goodenough, and Eaves 1985; St. Dennis 1986). Of these, St. Dennis provides by far the most extensive set of guidelines for writing reusable Ada source code. Most of the guidelines in the following list are from St. Dennis—and, we have included most of St. Dennis' guidelines. The "Exceptions" guidelines, and a few others, are from ISEC. Most of the guidelines suggested by Basili et al. (1989) are included in the list.

The guidelines follow, grouped by Ada features and issues. St. Dennis relates these guidelines to his reusability characteristics (discussed in the previous section), showing how the characteristics are supported by the guidelines.

SUBPROGRAMS

- AS1: *Separate subprogram declarations and bodies for ease of recompilation and modification.*
- AS2: *All reusable subprograms except a main program must be written within a library unit package.*
- AS3: *Use subprogram declarations to specify interfaces to reusable objects. Use subprogram bodies to implement these interfaces and properties of the objects.*
- AS4: *Write subprogram interfaces at an appropriate abstract level.*
- AS5: *Standardize the format for first-level, package-nested subprogram declarations, including regions for purpose, parameter descriptions, and associated documentation.*
- AS6: *Standardize the format for secondary unit (subunit) and first-level, package-body-nested subprogram bodies, including regions for revision history, purpose, associated documentation, parameter description, assumptions/resources required, side effects, diagnostics, data declarations, packages, operations, and algorithmic code.*
- AS7: *Write subprogram bodies to handle interaction with/effects on their environment.*
- AS8: *Write subprogram bodies with one normal exit and a grouped set of abnormal exits via exception handlers.*
- AS9: *Write subprogram bodies that pass results back to callers rather than use results to effect their function.*
- AS10: *Exploit formal parameter modes to clarify subprogram interface semantics.*
- AS11: *Use named-parameter associations for calls on subprograms with more than three parameters or in any case for interface clarity.*

28 SOFTWARE REUSE GUIDELINES

AS12: *Minimize subprogram overloading.*

PACKAGES

- AP1:** *Write library unit package specifications and bodies in separate files for ease of recompilation and modification.*
- AP2:** *Use package specifications to specify the interface to object abstractions; use package bodies to encapsulate implementation-specific details of these abstractions not needed by client software.*
- AP3:** *Design packages to implement interfaces to reusable objects at a consistent abstract level.*
- AP4:** *Standardize the format for library unit package specifications, including various regions for revision history, purpose, associated documentation, diagnostics, packages, data declarations, operations, and private types.*
- AP5:** *Standardize the format for secondary unit package bodies, including regions for revision history, purpose, associated documentation, assumptions/resources required, side effects, diagnostics, packages, data declarations, and operations and initialization code.*
- AP6:** *Use private or limited private types and the private part of package specifications to restrict client software's view of data and operations on that data.*

VISIBILITY RULES

- AV1:** *Do not use USE context clauses.*
- AV2:** *Use renaming declarations to resolve name conflicts with the environment.*
- AV3:** *Use renaming declarations to facilitate modifying reusable software to represent new object abstractions.*
- AV4:** *Do not hide package standard (i.e., do not use the identifier STANDARD as a user-defined name).*

TASKS

- AT1:** *Separate task declarations and bodies for ease of recompilation and modification.*
- AT2:** *Use task declarations to specify interfaces to reusable objects. Use task bodies to implement these interfaces and properties of the objects.*
- AT3:** *Write task interfaces at an appropriate abstract level.*
- AT4:** *Standardize the format for first-level, package-nested-task declarations, including regions for purpose, entry descriptions, representation clause descriptions, and associated documentation.*
- AT5:** *Standardize the format for secondary unit (subunit) and first-level, package-body-nested-task bodies, including regions for revision history, purpose, associated documentation, assumptions/resources required, side*

effects, diagnostics, packages, data declarations, and operations and algorithmic code.

- AT6:** *Write task bodies to effectively handle interaction with/effects on their environment.*
- AT7:** *Write task bodies with one normal exit or termination point and a grouped set of abnormal exits via exception handlers.*
- AT8:** *Write task body accept statements that pass results back to callers of the task rather than use results to effect task function.*
- AT9:** *Use task types to define reusable operations on data and task objects to implement particular (distinct) instances of these operations.*
- AT10:** *Exploit entry formal parameter modes to clarify task interface semantics.*
- AT11:** *Group all default parameters in entry parameter specifications at the end of the specifications.*
- AT12:** *Use named parameter associations for calls to task entries with greater than three parameters or in any case for interface clarity.*
- AT13:** *Minimize entry overloading.*
- AT14:** *Write all select statements with an else part or include a handler for the PROGRAM_ERROR exception at the end of the enclosing task block.*
- AT15:** *Minimize use of task priorities or modify priorities accordingly when composing tasks with other tasks for the sake of reuse.*
- AT16:** *Minimize use of abort statements.*

PROGRAM STRUCTURE AND COMPILATION ISSUES

- APS1:** *Use library unit package specifications as the encapsulation mechanisms for directly reusable software (i.e., data and operations on the data).*
- APS2:** *Allow only first-level, nested-nonpackage entities in library unit package specifications to form the basis for catalogued, directly reusable objects/software.*
- APS3:** *Use secondary unit package bodies, package specifications containing only data, and subunits corresponding to first-level, package-body-nested stubs as the encapsulation mechanisms for indirectly reusable software.*
- APS4:** *Use WITH clauses on package specifications to reference only data needed in specifications. WITH clauses can be used freely on package bodies as needed.*
- APS5:** *Use subunits to achieve modularity and ease of recompilation.*
- APS6:** *Do not split an abstraction across several packages. Use layers of packages to reflect the abstraction.*
- APS7:** *Use separate compilation and separate specifications and bodies to achieve modularity and ease of recompilation.*
- APS8:** *Implement bodies of subprograms declared in a package specification as subunits of the package body.*

30 SOFTWARE REUSE GUIDELINES

APS9: *Minimize use of Pragma Elaborate.*

GENERIC UNITS

- AG1:** *Use generic program units (i.e., packages and subprograms) to parameterize reusable software parts.*
- AG2:** *Use generic program units to specify precise module interfaces/imports and exports.*
- AG3:** *Use generics to allow specification of multiple instances of reusable software as compared to reuse of one shared instance.*
- AG4:** *Use base types rather than subtypes to specify the type of a generic formal object or generic formal subprogram parameter or to specify result types.*
- AG5:** *Standardize the format for Library unit and first-level, package-nested, generic unit declarations, including a region for description of generic parameters as well as standard information required for nongeneric subprogram and package declarations.*
- AG6:** *Separate generic declarations from bodies for ease of recompilation and modification.*
- AG7:** *Exploit generic formal object parameter modes to clarify interface semantics.*
- AG8:** *Use generic type definitions to clarify interface semantics and module operation.*
- AG9:** *Use additional generic parameters as necessary to effect inheritance of desired operators on generic formal types.*
- AG10:** *Minimize generic formal subprogram parameter overloading and overloading of subprograms in generic packages.*
- AG11:** *Minimize use of the box notation (is <>) to specify default generic formal subprograms as parameters.*
- AG12:** *Use basic operations/attributes associated with generic formal types to provide required generality to generic bodies.*
- AG13:** *Use named-parameter association in actual parameter parts of generic instantiations.*
- AG14:** *Use default parameters for generic, actual parameters whenever possible.*
- AG15:** *Create particular instantiations of generic units corresponding to common uses of reusable software.*

EXCEPTIONS

- AE1:** *For each assumption a subroutine depends on to operate correctly, define an exception that is to be raised when the assumption is violated.*
- AE2:** *For every situation that would raise an exception, define a function that indicates whether the exception would be raised.*
- AE3:** *Raise an exception if the user can easily fix the problem. Let the user provide a subprogram to fix the problem if raising an exception causes loss of information.*

AE4: *Provide a subprogram to return all available information describing the nature of an exception situation.*

St. Dennis (1986) offers detailed discussions and examples for the guidelines obtained from that document. The reader is encouraged to consult St. Dennis and ISEC (1985) for further information.

The following guidelines briefly summarize our observations.

CC16: *Emphasize Ada as a suitable programming language for reusable code.*

CC17: *Establish a set of organizational guidelines for code development.*

3.2.4 Component Quality

If software reuse is to be successful, the available reusable components must be worthy of user confidence. Numerous aspects bear upon the achievement and maintenance of component quality, including the understandability and completeness of requirements and designs, the quality of documentation, programming style, and ultimately the correct performance of the reused software. Two underlying issues that require special attention are component validation and verification (V&V) and configuration management.

The importance of the configuration management issue is well known to software engineers. Mechanisms for configuration management are practiced effectively by many organizations. Since having the trust and confidence of reusers is critical, great emphasis should be placed on workable configuration management procedures.

As we noted in Chapter 2, there is typically some resistance to using software created outside an organization. Every effort must be made to overcome any legitimate concerns about the quality and reliability of such software. Another facet of this issue is the potential liability the creating organization may have. In most respects this is an unresolved legal issue (Chapter 2); but at a minimum the creating organization's reputation and business potential are affected by the perception of users of its products.

The issue of component V&V is especially significant for reusable software. Bullard et al. (1989) have studied this issue under the AIRMICS Ada Reuse and Metrics project. As they observe, in the past the focus of V&V was to determine whether software met its specification within a specific environment. In the case of reusable software, however, the software may be reused in numerous different environments. They differentiate between the "machine environment" (physical hardware and virtual machine incorporating operating system and any run-time environment and conventions set up by the compiler) and "application environment" (application-programmer-written components with which the reusable software component is to be used). They note that "porting" a component is changing its machine environment and "adapting" a component is changing its application environment. This gives rise to the terminology of "portability errors" and "adaptability errors," both of which must be avoided with reusable software. Portability and adaptability effectively define two different interfaces: a "machine interface" and an "application interface." Reuse errors may be usefully considered relative to these interfaces.

Bullard et al. (1989) state that portability errors are contained entirely within the reusable component and can be detected during unit testing (as opposed to integration testing). Adaptability errors, on the other hand, span the application interface and represent only potential errors in the component, since it is impossible to anticipate all potential application environments—i.e., application interfaces. Adaptability problems thus must be considered at systems integration

time—although the potential for an adaptability error may be detectable by considering the component alone. The Ada language definition considers components with portability errors to be erroneous (Bullard et al.). Many such errors can be detected statically, since language constructs that permit some compiler or operating system latitude are known. On the other hand, adaptability errors are not amenable to formal verification techniques, since any error is relative to a specific application environment. Whereas portability has been extensively studied and is relatively well understood (including problems with specific programming languages), adaptability errors are much more difficult to resolve.

Bullard et al. (1989) give a good discussion of approaches for detecting reuse errors; in brief, they include (1) testing a reusable module in a variety of actual environments (which cannot be all-encompassing), (2) testing the module with simulated application environments, (3) considering static measures of portability and adaptability, (4) conducting mutation analysis, and (5) constraint-based analysis.

Bullard et al. (1989) suggest three general categories for portability errors; they also suggest using these categories to recognize or to detect portability errors and to remove or adequately document them.

CLASSIFICATION OF PORTABILITY ERRORS

(a) Types of Application

- (i) numerical (e.g., errors related to convergence, error propagation, floating point precision, overflow treatment, or byte alignment)
- (ii) multitasking (e.g., errors related to incompatibility of code from different compilers, timing and synchronizing problems, shared variable use, or use of machine priorities)
- (iii) dynamic memory allocation (e.g., problems due to space overflow, machine differences in available memory, or method of allocating pointers)
- (iv) systems programming (e.g., absolute location specification, object representation, maximum object space allocation, or unchecked conversions)

(b) Manner of Error Detection

- (i) static analysis (e.g., unchecked deallocation, machine code insertions, use of machine constants, or dependency on order of evaluation or on a particular memory initialization policy)
- (ii) simulation (e.g., method of interacting with the environment at execution time—allocating and laying out storage, passing parameters, elaborating library units, or synchronization of multiple processors)
- (iii) mutation analysis (e.g., problems detectable through manipulation of input, whether an exception is raised on overflow, or whether a machine rounds or truncates unrepresentable values)
- (iv) constraint analysis (e.g., the need for at least a certain precision for floating point calculations or the need for very large amount of memory for dynamic allocation)

(c) Language Constructs

Bullard et al. (1989) cite references to the literature concerning portability problems associated with specific Ada constructs.

The following categorization of adaptability is also from Bullard et al. (1989).

CLASSIFICATION OF ADAPTABILITY ERRORS

- (a) Architectural Concerns (Bullard et al. from Booch 1987)
 - (i) concurrency (sequential, guarded, concurrent, or multiple)
 - (ii) space utilization (unbounded or bounded)
 - (iii) space reclamation (unmanaged, managed, or controlled)
 - (iv) iterator availability (whether an operation is available to yield the elements of an object of an abstract data type)
- (b) Implementation Concerns
 - (i) order of evaluation errors
 - (ii) aliasing errors
 - (iii) domain errors (e.g., a subprogram defined on only values within a certain range)
 - (iv) protocol errors (e.g., a component within a retained state, subject to constraints—such as no more POPs than PUSHes for a stack component)

Bullard et al. note that, in the case of implementation concerns (item b), the common concern is that the component is designed to allow a client application to misuse the component, thus the potential for an error exists. They note that the realization of the error exists somewhere across the application interface in the application environment itself, otherwise the error is either a portability error (as in the case of aliasing, order of evaluation, etc.) or a traditional error (as in the case of a domain error). Thus traditional V&V techniques are of little use in detecting the problems.

The work reported here from Bullard et al. (1989) points up the difficulty in determining reuse errors—especially adaptability errors; certifying a component as error free relative to any arbitrary environment is impossible for the general case. Thus the goal should be to develop components with portability and adaptability considerations built in—by means of explicit specification of portability and adaptability requirements, isolating portability-related code, and building defensive assertions into the code relative to adaptability.

Another important aspect of software quality assessment is the recorded experiences of users of library components. We consider this in Section 3.3. Quality assurance activities are considered essential in organizations practicing software engineering. Due to the critical importance of the correctness of reusable software, even greater emphasis should be placed on the scrutiny of reusable components by a quality assurance group. ISEC (1985) suggests the use of a “reusability checklist” by the quality assurance group.

We conclude this section with the following guidelines, prefaced either with “Q” (for quality) or “V” (for V&V). The V&V guidelines were adapted from Bullard et al. (1989).

- Q1:** *Set standards to be met by all library components.*
- Q2:** *Emphasize stringent V&V for reusable components, stressing portability and adaptability.*
- Q3:** *Emphasize enforcement of standards and practices by the quality assurance group; employ a reusability checklist.*

34 SOFTWARE REUSE GUIDELINES

- Q4:** *Establish and operate an effective configuration management program for the reuse library.*
- V1:** *State environment compatibility explicitly in the requirements specification.*
- V2:** *Specify constraints on the use of reusable components as assertions; include assertions within the component specification and (if practical) within the executable code.*
- V3:** *Construct code for portability and adaptability (rather than attempting to isolate inadequacies by testing).*
- V4:** *Parameterize specifications that are dependent on the machine environment so that the behavior of the component is expressed relative to a part of the machine environment.*
- V5:** *Classify reusable components along each of the dimensions of concurrency, space utilization, space reclamation, and iterator availability; make the classification a part of the component specification.*
- V6:** *Use suggested classification of reuse errors to recognize or to detect errors automatically; remove errors, if possible, or adequately document their existence for location and modification when necessary.*
- V7:** *Employ a comprehensive testing method, consisting of a combination of approaches, to detect reuse errors (including simulation of the execution environment, static analysis, mutation analysis, and constraint-based analysis).*
- V8:** *Include adequacy criteria that reduce the amount of testing needed and that provide a measure of a test suite's effectiveness.*

3.2.5 Classifying and Storing Components

Having determined that components will be made available for reuse, it is necessary to classify each component according to some taxonomy and to store the component in an on-line library for reuse. An organization must make decisions early-on about the scheme(s) for classification, and carefully classify each component for reuse. The decision has many ramifications, since the approach for locating and retrieving available components (Section 3.3) is largely determined by the approach for classifying and storing.

Considerable research and experimentation have been conducted in this aspect of reuse repository operation. Prieto-Diaz and Freeman (1987) have developed the faceted classification method, based on ideas from library science. Each component is characterized by a sextuple consisting of:

<function, objects, medium, system type, functional area, setting>.

They incorporate the idea of conceptual closeness, to give a user a measure of how closely an available component corresponds to a specified facet during retrieval.

Ramamoorthy et al. (1986) have selected the entity-relationship-attribute (ERA) model as the basis for cataloging and retrieving reusable components. They use such attributes as classification (requirements, design, source, test case, document, library, and object code); hierarchy (family, member, layer, module, and procedure); and nonfunctional attributes such as reliability, memory requirements, performance, and metrics concerning quality and complexity. Their relations include classification-to-classification, hierarchical_level-to-hierarchical_level,

use of resources, etc. They have developed the Entity Specification Language (ESL) to support software personnel in inserting, modifying, or deleting information in the library.

Mittermeir and Rossak (1987) have proposed library organizations that they call software archives (to support the retrieval of design units for potential reuse) and software bases (to support retrieval of executable code). They represent the structure of a software archive by a four-dimensional "cube," the dimensions being decomposition, representational form, association (application-dependent links), and generalization/specialization. This structure thus uses links among components to represent knowledge about interrelationships.

Wood and Sommerville (1988) have taken a cataloging approach based on natural language processing—specifically the idea of concept case frames. They rely on a system creating component descriptor frames (their version of concept case frames) by means of a form-filling interface. There is a component descriptor frame for each basic function that software performs—representing a class of conceptually similar verbs (e.g., search, look, and find). There are four slots in each frame corresponding to a library component, objects manipulated by the component, objects produced as a result of the action, and objects that provide a context for the action. An example is (in order) a report generator, a personnel record, a formatted report, and directives that describe the desired report. This organization was devised to improve the process of retrieval (Section 3.3.1).

The RAPID Center Project was discussed previously (in Section 1.4). As part of this project (being pursued by SofTech under contract to the U.S. Army ISEC) the RCL System is being developed (Guerrieri 1988). Classification is based on the faceted classification scheme of Prieto-Diaz and Freeman (1987). User activities supported relative to reusable software components are identification, extraction, and report generation. The RAPID Center will also provide guidance to users by technical staff.

Gagliano, Fraser, and Owen (1989) have developed an experimental reuse library system, based in part on Prieto-Diaz's work, for reusable Ada components. They are developing several tools to ease the development and use of a reuse library.

For effective retrieval, the library should represent relationship information between components (e.g., between a design component and the corresponding code, between a system specification and a subsystem specification, between two components that are related by reuse potential within a given application domain, and between two components such that one is a specialization of the other). Research has been conducted to examine appropriate structures for database support for software reuse—i.e., to determine how best to use the knowledge and systems of database management and information retrieval to support the creation and use of software repositories. Bein, Drew, and King (1989) suggest the use of both the object-oriented data model and the semantic data model in creating and managing libraries for software reuse. They cite the need for composite objects, multiple inheritance, representation of relationships (one-to-one, one-to-many, and many-to-many), and monitoring of data integrity. Relative to data base functionality, they cite the need for dynamic schema evolution, transparent operation within distributed environments, version control, security and authorization support, and transaction management and concurrency control.

Frakes and Nejme (1987) take an entirely different approach to the structure of reuse libraries, arguing that with the advent of fast special-purpose hardware for information retrieval (IR), it is feasible to use IR approaches for textual-based

36 SOFTWARE REUSE GUIDELINES

components search and retrieval, rather than use elaborate database organizations. We will consider these ideas further in Section 3.3.

Regarding the question of what documentation should be provided on-line with each reusable component, Van Scoy and Plinta (1989) suggest two levels of information for code components:

1. high-level description of the functionality provided by the component, and
2. detailed description of the components.

The high-level description is assumed to describe the component's functionality in terms familiar to prospective users. The approach of domain-specific libraries lends itself to this end. The detailed description should include the following seven items. Items 1 through 6 are from Van Scoy and Plinta (1989) and item 7 is from Plinta (1989):

1. what it needs,
2. what it provides,
3. what it does (not to be confused with how it does it),
4. performance documentation,
5. rationale-type documentation,
6. test software and documentation, and
7. adaptation description.

Please see the papers for discussions of these items. We include some suggestions for Ada code component documentation in Section 3.2.3. The STARS *Reusability Guidebook* (Wald 1986) suggests the following documentation to accompany a submittal to a library.

Part description: title, type of component (code, design, etc.), type of function, purpose of function, interface requirements (include required information about the software and hardware not included with the component)

Submitter data: name, address/network address, phone, contact

Component constituents: abstract, requirement specification, functional specification, design, algorithm or function, source code, object code, test specification, test code, test data/results, maintenance/operations (user's manuals), training materials

Component history: reason for component development, date of completion, description of applications used, frequency of use, description of development standards, version number

Component relationships: name of parent, name of children, name and version of siblings, used by, uses

Component attributes: keywords (to search/retrieve on), development language, host environment (computer and operating system), target environment (computer and operating system)

Restrictions: government, developer, environment imposed (e.g., compiler, tools, peripherals), reusability metrics

Disclaimers: warnings, problems, limitations, lack of tests

Software support: support organization or person, qualification, frequency of update

Miscellaneous instructions: how to get part, fees, warranties

Releases: transfer and/or assignment of copyright, transfer of ownership of hard goods

Deliverable media description: media, hard goods in delivery

Media: electronic, magnetic, optical, paper, type of format (e.g., ASCII record, etc.)

As we have noted, code components should not be retained in the library without the corresponding design, requirements, and test components. The relationships between them must either be explicitly stated (in the descriptions for the components) or be inherent in the library organization. Frakes and Nejme (1987) make specific suggestions about the form of structured comments to include with code, as do Chen and Ramamoorthy (1986). The structured-comments approach lends itself to automated extraction of the information fields for on-line retention in the library. Biggerstaff (1989), of MCC, emphasizes the use of hypertext for depicting relationships and interconnecting available documentation for components.

We end the section with the following guidelines.

- CC18: *Determine approach(es) for classifying and storing components based, e.g., on domain analysis.*
- CC19: *Represent relationships between a component and its more (and less) abstract representations (as to life-cycle phase).*
- CC20: *Represent relationships between a component and others that may collectively solve a given problem (or class of problems).*
- CC21: *Document each component thoroughly on-line, including user documentation and programming (i.e., maintenance) documentation.*

3.3 REUSING COMPONENTS

A number of operational issues must be satisfactorily addressed to effectively make use of available reusable components. Included are (a) classifying and storing components (which we considered in Section 3.2.5), (b) identifying components that meet specific needs, (c) understanding identified components, (d) adapting components as necessary, and (e) assembling components into a complete software system. In this section we consider issues (b) through (e).

3.3.1 Searching and Retrieving

In the case of the faceted classification scheme (Prieto-Diaz and Freeman 1987), the search is based on specification of a sextuple of descriptive keywords, or facets (see Section 3.2.5). The user forms a query using the thesaurus to select a representative term for each facet. The user can use the asterisk rather than a keyword for a facet to generalize the query. The user can also request "expansion" of a query by having one of the specified facets replaced by other keywords, in order of conceptual closeness. The researchers have also attempted to provide ranking of the retrieved components based on user profiles; e.g., it is more important for novice programmers to attempt to use very small programs (in terms of lines of code) than is true for more experienced programmers.

Ramamoorthy et al. (1986) have developed the Resource Extractor (REX) for use in forming queries using attributes and/or relations as qualifiers. Examples of their queries are:

```
SET   High_Req = Software_Resources
      Classification = Requirements
      Performance = High.
```

This results in the creation of a set `High_Req` containing those elements of the set `Software_Resources` whose elements satisfy the specified relation and attribute.

Wood and Sommerville (1988) provide a forms-based interface to their library implementation on a Sun workstation; it prompts the user for either a verb describing the action of a component or for a noun representing an object manipulated by the component. Thereby the system undertakes construction of a component descriptor frame (Section 3.2.5); partially-completed frames are used to search the database; the user is provided lists of candidates for filling in other slots; and on-line helps are available. The user can select either a keyword search, in which an exact match on specified names must occur, or can permit components to be retrieved based on conceptual classes of verbs, for example. They plan to extend the system to include a browsing mode, and tools to integrate reuse with automated design tools.

Tarumi et al. (1988) have developed a rule-based retrieval mechanism, based on user inputs of object names, attributes, relations, and operations. Names may have aliases. Their approach is a mixture of formal and informal methods; they emphasize the importance of this combination, believing that it yields simplicity and user friendliness.

Frakes and Nejme (1987) propose the use of IR systems for locating code components for reuse (and this could be extended to other abstraction levels). IR systems deal with formatted text, as well as unformatted text, which is not usually dealt with by database management systems. They have devised a novice user mode, which is menu driven, and a command mode for more experienced users. They have developed an experimental system called CATALOG, written in C under UNIX and MS-DOS. Help messages are available, and partial term matching is done using such sophisticated IR techniques as automatic stemming and phonetic matching. They propose that all components submitted to a reuse library must begin with a standard prologue of descriptive information, to form the basis for subsequent search and retrieval of the component (e.g., name, description, supporting documents, author, date, usage, parameters). With the advent of special-purpose hardware for IR (e.g., see Smith 1989) and resulting greatly improved responsiveness, IR techniques should be considered as candidates for reuse library support. Frakes and Nejme (1987) also mention the promise of user feedback in conjunction with IR use.

Three important aspects of the IR process are discussed by Wood and Sommerville (1988). They are recall, precision, and ranking. Recall pertains to the percentage of relevant components that are identified; precision pertains to the percentage of identified components that are relevant; and ranking orders identified components by quality of match (to address the "information overload" issue). These are all substantive issues, and will be increasingly important as reuse libraries become larger.

In Section 1.4 we summarized a successful reuse project reported by Carstensen (1987). He described an approach to library browsing, in which the user may specify English language nouns and verbs as search vectors. Based on potential matches, the user is first presented two-sentence abstracts of all candidate components. If

desired, a more complete abstract (up to one page in length) may be requested for any of the components.

Several of the retrieval approaches mentioned above emphasize user-friendly interfaces. Burton et al. (1987) discuss and depict their approach to an interactive interface in their RSL system, supporting all aspects of reuse. The ease of use of the reuse system is a very important issue, and deserves special emphasis in planning for and implementing reuse within an organization.

The following guidelines pertain to the above discussion.

RC1: *Devise and implement a mechanism for search and retrieval supporting query and browsing modes.*

RC2: *Emphasize user-friendly interface for search and retrieval,*

RC3: *Provide indication of "goodness of match" of components to a query.*

3.3.2 Understanding and Assessing Components

Standish (1984) estimates that software maintenance costs 70 to 90 percent of the life cycle, and understanding accounts for 50 to 90 percent of maintenance cost. This would mean that understanding accounts for 35 to 80 percent of life cycle cost. Understanding is absolutely critical to software reuse—especially if a component must be adapted.

There are many aspects to component understanding, including the creation of the component within the framework of a well-understood, consistent software engineering process (Section 2.2), and good approaches to development, testing and maintenance, as discussed in Section 3.2. The existence of effective domain analysis, and knowledge of how a component fits within the domain, can be most helpful in understanding how the component may be used. In short, if good practices are followed in developing, classifying, and storing reusable components, thorough understanding should be a natural by-product.

Bott and Wallis (1988) argue that (a) we need to use components that implement fairly complicated functions to achieve large benefits from reuse, and that (b) to do so it is essential to reduce the perceived complexity of components as seen by the system designer. They maintain that, to this end, components must be designed for reuse from the beginning; and, their major theme is that components must conform to some kind of simplified "user model" of the system that they support, to relieve a user of detailed coding concerns. An example of such a user model is that of compiler construction (the front-end/back-end division, and the compiler phases). They also note that UNIX users benefit from the simplicity of its "user model"; although it is a very complicated system, enough can be learned very quickly to allow useful activities to occur. Biggerstaff and Richter (1987) refer to this simplified model as the "mental model," and state that developing such a model is probably the fundamental operational problem to solve in development of any reuse system. They suggest the use of hypertext to help solve the problem.

We often face the need to understand and seek to reuse existing software that was not developed for reuse, and is not well understood by prospective reusers (nor perhaps by anyone available to the prospective reusers). Some approaches are discussed here for dealing with this issue. Chen and Ramamoorthy (1986) have developed the C Information Abstractor, which scans C programs and stores information into a database. The information obtained primarily relates to objects that can be accessed across C file or function boundaries—namely, files, functions, global variables, global types, and macros. An Information Viewer is provided to operate on the database, and provide answers to such questions as: what functions call a given

function, where is a certain variable defined, what functions access a given global variable, and what is the type of a variable. Since some important information cannot be extracted from the code (e.g., underlying assumptions, the algorithm used, computational complexity, necessary preconditions), the authors propose the use of structured comments to provide the information to the Abstractor. Examples of such information they suggest are: purpose, assumptions, preconditions, assertions, algorithm description, algorithm complexity.

Chen and Ramamoorthy (1986) also comment on software metrics, noting that software quality, testing required, and maintainability, depend on such metrics as function-to-file bindings, file-to-file bindings, the number of objects related to a given function, and the number and depth of calling paths starting from a function. They observe that an examination of such metrics may well indicate the need for restructuring prior to reuse. They are considering means to handle some of the details automatically, using the program database.

Sayani's (1989) research in "reverse engineering"—approaches for "unraveling the product ... to its earlier life cycle development phase(s)"—can provide important leverage in aiding the understanding process and alleviating to some extent the need to attempt to understand code.

Biggerstaff (1989) describes work at MCC in reverse engineering, providing a good overview of their reverse engineering system (Desire Version 1.0), and summarizing other commercially available reverse engineering tools. Desire operates within the framework of an object-oriented domain model, and consists of a parser, a set of post-processing functions, and the PlaneText hypertext system for use in presentation. There is considerable use of informal knowledge in their process, as a human user interacts with and guides the process.

DeMillo et al. (1989) at Purdue emphasize the use of operational history information in assessing candidate reusable components. They developed an experimental system to log some types of observations on component use, for query by a prospective user. They suggest the use of such experience information as security observations, extent of use, reported failures and faults, and performance observations (e.g., execution efficiency, memory utilization). Some of this data can be collected automatically, while some must be supplied explicitly by users.

The importance of accumulating such usage data should be stressed, and an organization should implement procedures to acquire the necessary feedback. Now we offer summary guidelines for this section.

- RC4: *Seek to understand reusable components chiefly through effective domain analysis, good software development practices, and good classification and storage mechanisms.*
- RC5: *Seek approaches/tools to help understand software not specifically prepared for reuse (e.g., reverse engineering).*
- RC6: *Use operational history of components in assessing their suitability for reuse.*
- RC7: *Obtain feedback from users of components, including number of uses, degree of satisfaction, and errors.*

3.3.3 Adapting Components

The ideal situation is that a component (or components) will be identified that exactly meets the need. That will often not be the case, however. Understanding the component (Section 3.3.2) is the key to the decision process. "Goodness of

fit" of an available component might well be measured by the effort required for adaptation. This would provide guidance when multiple components are identified that are candidates for selection. If a code component is functionally adequate—i.e., it performs a needed role in an acceptable way—then there should be little or no adaptation required provided that the component is highly cohesive and has no side effects. Ramamoorthy et al. (1986) give some parameters for use in deciding whether to reuse a component. They are:

Nnr, number of lines to be written if no reuse;
Nmr, number of lines to be modified for reuse;
Nr, number of lines that are being reused;
Enl, effort the organization needs for writing a new line;
Eml, effort the organization predicts for modifying a line;
Emal, effort the organization needs for maintaining a line;
Nrb, number of times previously reused;
Net, number of errors found in some fixed period;
Nel, number of errors found per fixed number of lines;
DocQ, documentation quality;
DesQ, design quality;
M, match for non-functional characteristics;
Exp, experience with the package to be reused;
Org, organization maintaining the package to be reused;
Dev, availability of the original developers; and
Time, time available for the project.

Some suggestions for deciding which component to reuse are:

1. The inequality $Nmr \times Eml < Nnr \times Enl$ should be true.
2. *Net* and *Nel* should be less than some predefined constant.
3. *DocQ* and *DesQ* should be greater than some predefined minimum constants.
4. Make use of *Nrb*, *Net*, and *Nel* in assessing the quality of a candidate component.

Space limitations prohibit discussing these ideas in more detail here; they are included as a beginning point for considering such issues. The interested reader may see the referenced paper for more background and details.

If code requires adaptation, the design and/or specifications corresponding to the code component (and hopefully retained in the library) will likely prove to be very important. If the component must be rewritten in a different programming language, the high-level design should serve as the basis—which is also true if the number of code "patches" required for adaptation is large. It is apparent that a high-level design should be easier to adapt than code, and that a specification is more general and more adaptable than a design.

Parameterized code is developed with the intent that input parameters cause "adaptation" of the code, as pre-planned. And, generators are driven by input directives to "adapt" within a pre-planned range. Ada generic procedures provide a mechanism for developing a "family" of procedures for which data types may be specified—and thus a specific "adaptation" created. It is very likely, at least in the near term, that most adaptation will be done manually. To that end, as suggested earlier, it will be extremely beneficial if the developers of a component will record suggestions for adapting the component for somewhat different uses.

Basili et al. (1989) are undertaking research to evaluate the reuse implications of Ada modules based on the explicit and implicit bindings of the module with its environment. They are experimenting with tools to assess the effort necessary to transform Ada code for reuse in different contexts. Some research is addressing the desire to build "automatic adaptation" capability into code. Two examples of such research follow.

Asdjodi has developed, as part of a prototype reuse system, the capability to automatically alter data structures as required for use by a selected component; thus, for example, if the output of one component is a matrix and the input for another is a linked list, her knowledge-based system would cause automatic generation of the required linked list (Asdjodi 1988, Asdjodi and Hooper 1989). In the prototype system only matrices and linked lists are used; however, the concepts could be extended to any type of data structures. She uses the very high-level language ELL as a means for specifying components to be composed; the user is thus shielded from many details.

Notkin and Griswold (1988) have developed a UNIX-based "extension" mechanism, based on an Extension Interpreter (EI). The EI consists of an arbitrator, which hierarchically maps procedure names to procedure implementations; the dynamic linker, which gives the flavor of interpretive environments like LISP; and the translation subsystem, which translates data between representations used by different languages. These components connect program components with a user interface. Their emphasis is on reusing source code without the need to change it. Thus, the more fine-grained the available procedures, the more likely that new capabilities can make use of them.

The following guidelines are provided.

RC8: *Use higher-abstraction representations in adapting a component (e.g., use design when adapting code).*

RC9: *Emphasize the use of available metrics/tools to assess adaptation effort.*

3.3.4 Composition of Code Components

"Composition" refers to interconnecting components to form software systems. The most straightforward approach may be used when the component is a procedure that perfectly meets the need; then composition results from a procedure call, coupled with the action of the "linker." The same is true, of course, if we can successfully adapt a procedure. This is at present the primary mechanism for composition.

Other important mechanisms for composition are UNIX pipes, and inheritance in object-oriented languages. Both of these have considerable benefit in shielding the user from the need to understand code, per se; in the best case the code can be treated as a "black box."

Tracz (1987a) describes a reuse system based on Ada components, using both parameterization and application generators. He describes an interactive dialogue of menus and prompts to obtain necessary parameters for a particular application. Then, based on the component library and the parameters, the generator creates a compilable Ada application program. To prepare for reuse in a given application domain, it is necessary to do a domain analysis (discussed in Section 3.1), identifying likely candidate applications. Parameterizing must be done, ranging from something as simple as character strings that may be substituted in the source code, to specification of how to assemble pieces of a program. In the latter case, it could

be that an existing program was "dissected" for just this purpose, as a result of domain analysis.

Goguen (1986) proposes to achieve composition by means of the "library interconnection language" (LIL). As it stands, LIL's syntax is Ada-like, and relies on Goguen's earlier work based on specification by use of axioms. He lists the following desirable techniques for constructing new entities from old ones:

1. Set a constant (e.g., the maximum depth of a stack).
2. Substitute one entity for a stub or parameter in another.
3. Sew together two entities along a common interface.
4. Instantiate the parameters of a generic entity.
5. Enrich an existing entity with some new features.
6. Hide (abstract or encapsulate) some features of an existing entity, either data or control abstraction.
7. Slice an entity to eliminate unwanted functionality.
8. Implement one abstract entity using features provided by others (leading to the notion of a vertical hierarchy of entities).

LIL is an example of a "module interconnection language." The goal of such languages is to interconnect modules—which may be written in different programming languages—without having to modify the modules, assuming that they provide needed functionality.

In Section 2.2.3 the object-oriented MELD mechanism of Kaiser and Garlan (1987) is discussed. The idea is to compose object-oriented components by merging data structures and methods from different components.

The following guidelines ensue from the discussion of this section.

RC10: *Use existing mechanisms for composition to the extent possible (e.g., procedure linking, UNIX pipes, inheritance in object-oriented languages, etc.).*

RC11: *Seek automated approaches to composition as understanding permits.*

3.4 TOOLS AND ENVIRONMENTS

Software reuse should be practiced as an integral activity within a software engineering process (Section 2.2). Thus the tools and environments for software engineering are directly supportive of software reuse (including use and creation of reusable components). There are, however, some needs unique to the reuse process, per se. The newness of software reuse means that there are few mature tools and environments created especially for software reuse. As far as is known, no commercially-available system presently purports to cover all the requirements for software reuse.

Following the 1987 Minnowbrook Workshop (Rogerson and Bailin 1987), a report was written summarizing the work of the working groups (Agresti and McGarry 1988). Here we will briefly consider the findings of Working Group 3 (Tools and Environments for Software Reuse). Clearly these tools are not all necessary to conduct software reuse, but are important as research goals to help automate the reuse process.

The working group participants determined the following six categories of reuse-related activities that need to be supported: domain analysis, development and

44 SOFTWARE REUSE GUIDELINES

refinement of reusable products, classification, searching, evaluation and assessment, and incorporation of reusable items. Of this list, the first three categories pertain to the creation of reusable components, and the last three to component reuse. We will consider their recommendations for the six categories, in order.

1. Domain Analysis

Key subactivities: knowledge extraction, identification of objects and operations, abstraction and relationships, classification and taxonomy, and domain languages and synthesis.

Support tools: [characterization codes: M (mature), D (developed but unproven), U (undeveloped)]

- * Knowledge extraction tools, such as expert system building tools (D)
- * Entity-relationship diagramming tools (D)
- * Object-oriented development tools (U)
- * Semantic clustering and automatic classification tools (U)
- * Computer-aided software engineering (CASE) tools (D)
- * Parsing tools (M)

2. Development and Refinement of Reusable Products

Key subactivities with supporting tools:

- * Isolate replaceable features: object-oriented languages (D), historical data collection (M)
- * Make products self contained: dependency analyzers (e.g., cross-reference generators) (D); structure analyzers (D)
- * Parameterize: software and language features such as table-driven software, macro expansion, preprocessors, and generics (D)
- * Enrich (add features to a product to widen its range of application): object-oriented languages (D), configuration management tools to manage multiple versions of a product (M)
- * Abstract and specialize: languages that support class hierarchies and inheritance (e.g., Smalltalk) (M)
- * Test and validate (to mitigate risk in reuse—the not-invented-here syndrome): test coverage analyzers (M)
- * Formally verify: verification environment (U)
- * Assess quality: quality metrics (D), standards checkers (M)
- * Restructure (for multilevel reusability): reusability metrics (U)

3. Classification

Approaches: controlled vocabulary (e.g., faceted classification approach), uncontrolled vocabulary (e.g., Frakes and Nejmeh 1987), and knowledge representation.

Support tools: context clarification tool (D), semantic closeness tool (D), thesaurus construction tool (D), boolean IR system (D), vector space

IR system (D), word processing tools (M), semantic net shell (D), frame shell (D), and rule-based expert system shell (D).

4. Searching

Alternative approaches and support tools:

- * Natural language: CATALOG tool (Frakes and Nejme 1987) (D)
- * Structured queries: CATALOG tool (Frakes and Nejme 1987) (D)
- * Browsing: Hypertext (D)
- * Hierarchical: IMS, Smalltalk (D)
- * Semantic search: Automated Library System (ALS) (D), SEMANTX (D)
- * Citation search: Alicia (RADC) (D)

5. Evaluation and Assessment

Support tools: standard test sets (U), reuse-level measurement tools (U), attitude measurement tools (U), usage measurement tools (D), SMART environment (D).

6. Incorporation of Reusable Items

Subactivities and support tools:

- * Selection of variants: source code difference tools (M)
- * Instantiation: compilers (M)
- * Provision of data: data generation (D), forms management (M)
- * Template completion: prompters (M), macro expanders (M)
- * Modification safety: maintenance support (M)
- * Integration of items: linkers (M), smart editors (D), environments with integration paradigm (e.g., UNIX shell, Common Lisp, Smalltalk) (M)

Please see Agresti and McGarry (1988) for references to the above items that are not included here.

We have noted in previous sections of this document various tools activities supported under the AIRMICS Reuse and Metrics project. Among these are the Ada Data Binding Tool (Basili et al. 1989), cost modeling tools (Fairley et al. 1989), operational history recording/extraction tool (DeMillo et al. 1989). Also, Georgia State University (GSU) considered tools needs in general, and developed an experimental reuse system, GSU RSC_MGT (i.e., Ada Reusable Software Component Management; Gagliano et al. 1989). Reverse engineering tools (e.g., Biggerstaff 1989) are also significant for reuse support. And, various other tools have been mentioned in previous sections.

As was mentioned earlier, software reuse can be practiced without these tools, but it is important that these tools be developed—and they will no doubt be appearing over the next few years.

4. GETTING STARTED

A software reuse program must provide significant economic benefit to an organization to receive enthusiastic support—and even to survive. Thus an organization should approach undertaking reuse in a realistic, pragmatic way. Sufficient commonality in current and anticipated software projects must exist to justify investment in reuse (for staff training, library development, etc.). Thus it will be necessary to assess whether the organization has one or more areas of emphasis in which requirements for similar software frequently recur (or are expected to recur). In the case of a corporation, a business decision should be made whether to invest in a reuse program for an emphasis area. Whether for a government organization or a corporation, the decision should rest on present and expected requirements for recurring software, predicting whether the benefit from reuse will be sufficient to justify the investment.

In concert with the assessment of reuse potential, an organization should determine its current assets for reuse. Initially, within certain selected emphasis areas, an inventory should be made of available software with good potential for reuse. This could include commercially-available software with proven benefit to the organization, as well as internal software. The software identified can serve as an initial set of components for reuse. Another very important asset for reuse is the availability of professional personnel with knowledge and experience in the emphasis area(s) (i.e., domain experts).

Another important consideration in initiating a reuse program is whether the organization is making use of effective software practices. It would be of little use to attempt a software reuse program without having in place a systematic, consistent process for software development and maintenance. SEI's process assessment procedure (Humphrey et al. 1987) is the best-known instrument for such assessments, but in the same vein, Fairley et al. (1989) has suggested a reuse assessment instrument with a questionnaire based on the style of the SEI assessment questionnaire. Shortcomings in the software process should be remedied for the inherent benefit and to improve the basis for reuse.

Before proceeding with a reuse program, it is critical that top-level management be convinced of the worthiness of the undertaking. The economic analysis and assets assessment suggested above should be an important factor in the management decision to proceed.

Having received top management's approval, at least to investigate the potential for reuse, a reasonable next step would be to undertake a pilot software project employing reuse. The project should be chosen very carefully, considering available domain expertise, existing software assets for the domain, and the likelihood that the project will benefit significantly from reuse. The pilot project will of necessity be based on near-term, comparatively well-understood requirements. If the results of the pilot project are encouraging, the project can lead to wider implementation of reuse practices within the organization. A phased, evolutionary approach can help develop needed reuse knowledge and skills, as well as convince technical and managerial personnel of the viability of reuse. Integration of reuse into the organization's software process will be an important experimental activity of the pilot project; the process should be expected to evolve as additional experience is gained.

We have suggested numerous guidelines in earlier chapters, many of which pertain to a "steady state" reuse program in a sizable organization. Certainly not all of the guidelines are applicable to a pilot project (e.g., many of the organizational

structure guidelines will not apply), or to a small organization, or a small segment of a larger organization.

As reuse experience builds, management must decide whether to propagate reuse practices to additional projects and organizational segments and the rate at which the propagation will occur. As reuse becomes more widespread, organizational procedures and policies should evolve, including a software engineering process incorporating the use and creation of reusable products, and provision of incentives for individuals and projects. The organizational infra-structure should be changed over time to support and foster reuse. It will be necessary from the very first pilot project to have a library of components (including, perhaps, some "pointers" to commercially-available products not currently in the library).

The phased approach advocated here should involve considerably lower risk than trying to implement all aspects of a full-fledged reuse program immediately. The organization should continually assess the effectiveness of its reuse process, seeking to adjust as necessary to deal with problems encountered, and to gain advantage from improved methods, tools, and policies.

We offer the following summary of the phased approach. The steps are stated somewhat in a suggested order of occurrence, recognizing that there are implied feedback loops and iterations and that the exact implications of each step must be determined by current status of this "bootstrapping" process.

1. Make an assessment of current software engineering practices and remedy major shortcomings.
2. Obtain top-level management support to undertake investigation of reuse feasibility.
3. Identify one or more application areas (if any) that are important to the organization's mission, are well understood, and have recurring similar software requirements.
4. Conduct an inventory of reusable assets for the identified application area(s).
5. Establish an initial library of reusable components.
6. Determine and conduct a pilot software project employing reuse.
7. Evaluate experience/success with reuse in the pilot project, present results to management and obtain a decision whether to proceed.
8. Expand reuse activities to additional application domains and organizational segments, as success and management approval warrant.
9. Conduct the following activities as part of the practice of software reuse (in the pilot project and in all succeeding reuse efforts):
 - * Institute management policies and practices to encourage reuse.
 - * Institute/carry out a software engineering process incorporating the creation and use of reusable products.
 - * Adjust the organizational structure and staffing as appropriate to support reuse.
 - * Implement/update library mechanisms.
 - * Perform domain analysis of selected domains and develop/acquire enough reusable components to conduct reuse for the domains.
 - * Continually assess the effectiveness of the reuse-based process, and adjust/augment it as appropriate.

REFERENCES

- Agresti, W., and F. McGarry. March 1988. *The Minnowbrook Workshop on Software Reuse: A Summary Report*, NASA/GSFC, Greenbelt, Md., Computer Sciences Corporation, Beltsville, Md.
- Aharonian, G. July 1989. "Working paper," in *Proceedings of the Reuse in Practice Workshop*, eds. J. Baldo and C. Braun, Software Engineering Institute, Pittsburgh, Penn.
- Ancoat 1988. *Proceedings of the Sixth National Conference on Ada Technology*. March 1988. Arlington, Va.
- Anderson, C. M., and D. G. McNicholl. 1985. "Reusable Software—A Mission Critical Case Study," p. 205 in *Proceedings of Compsac 85*.
- Asdjodi, M. 1988. "Knowledge-based Component Composition: An Approach to Software Reusability," Ph.D. dissertation, The University of Alabama in Huntsville, Huntsville, Ala.
- Asdjodi, M., and J. W. Hooper. June 1989. "An Environment for Software Reusability," pp. 48–53 in *Proceedings of the First International Conference on Software Engineering and Knowledge Engineering*, Skokie, Ill.
- Bailin, S. C. October 1987. "Informal Rigor: A Practical Approach to Software Reuse," in *Proceedings of the Workshop on Software Reuse*, eds. G. Booch and L. Williams, Rocky Mountain Inst. of Software Engineering, SEI, MCC, Software Productivity Consortium, Boulder, Colo.
- Bailin, S. C. May 1989. "An Object-Oriented Requirements Specification Method," *Communications of the ACM* 32(5), 608–23.
- Bailin, S. C., and J. M. Moore. December 1987. "A Software Reuse Environment," in *Software Engineering Workshop*, NASA/GSFC, Greenbelt, Md.
- Baker, B., and A. Deeds. July 1989. "Industrial Policy and Software Reuse: A Systems Approach," in *Proceedings of the Reuse in Practice Workshop*, eds. J. Baldo and C. Braun, Software Engineering Institute, Pittsburgh, Penn.
- Baldo, J., and C. Braun, eds. July 1989. *Proceedings of the Reuse in Practice Workshop*, Software Engineering Institute, Pittsburgh, Penn.
- Barnes, B., T. Durek, J. Gaffney, and A. Pyster. July 1987a. "Cost Models for Software Reuse," in *Proceedings of the Tenth Minnowbrook Workshop (1987, Software Reuse)*, Blue Mountain Lake, N.Y.
- Barnes, B., T. Durek, J. Gaffney, and A. Pyster. October 1987b. "A Framework and Economic Foundation for Software Reuse," in *Proceedings of the Workshop on Software Reuse*, eds. G. Booch and L. Williams, Rocky Mountain Inst. of Software Engineering, SEI, MCC, Software Productivity Consortium, Boulder, Colo.
- Barsotti, G., and M. Wilkinson. March 1987. "Reuseability—Not an Isolated Goal," pp. A1–A14 in *Proceedings of the Conference on Software Reusability and Maintainability*, The National Institute for Software Quality and Productivity, Inc., Tysons Corner, Va.

- Basili, V. R., H. D. Rombach, J. Bailey, A. Delis, and F. Farhat. March 1989. "Ada Reuse Metrics," pp. 11-29 in *Guidelines Document for Ada Reuse and Metrics (Draft)*, P. A. Lesslie, R. O. Chester, and M. F. Theofanos, eds., K/DSRD-54, Martin Marietta Energy Systems, Inc., Oak Ridge, Tenn., under contract to U.S. Army, AIRMICS.
- Basili, V. R., H. D. Rombach, J. Bailey, and B. G. Joo. July 1987. "Software Reuse: A Framework," in *Proceedings of the Tenth Minnowbrook Workshop (1987, Software Reuse)*, Blue Mountain Lake, N.Y.
- Baxter, I. D. October 1987. "Reusing Design Histories via Transformational Systems," in *Proceedings of the Workshop on Software Reuse*, eds. G. Booch and L. Williams, Rocky Mountain Inst. of Software Engineering, SEI, MCC, Software Productivity Consortium, Boulder, Colo.
- Bein, J., P. Drew, and R. King. March 1989. "Object-Oriented Data Base Tools to Support Software Engineering," pp. 95-110 in *Guidelines Document for Ada Reuse and Metrics (Draft)*, P. A. Lesslie, R. O. Chester, and M. F. Theofanos, eds., K/DSRD-54, Martin Marietta Energy Systems, Inc., Oak Ridge, Tenn., under contract to U.S. Army, AIRMICS.
- Beaver, E. W. July 1989. "Reuse Experiences/Enhancements," in *Proceedings of the Reuse in Practice Workshop*, eds. J. Baldo and C. Braun, Software Engineering Institute, Pittsburgh, Penn.
- Biggerstaff, T. J. October 1987. "Hypermedia as a Tool to Aid Large Scale Reuse," in *Proceedings of the Workshop on Software Reuse*, eds. G. Booch and L. Williams, Rocky Mountain Inst. of Software Engineering, SEI, MCC, Software Productivity Consortium, Boulder, Colo.
- Biggerstaff, T. J. July 1989. "Design Recovery for Maintenance and Reuse," *Computer* 22(7), 36-49.
- Biggerstaff, T. J., and A. J. Perlis, eds. September 1984. "Special Issue on Software Reusability," *IEEE Trans. on Software Engr* SE10(5).
- Biggerstaff, T. J., and A. J. Perlis, eds. 1989a. *Software Reusability. Concepts and Models*, Volume I, ACM Press, Addison-Wesley, Reading, Mass.
- Biggerstaff, T. J., and A. J. Perlis, eds. 1989b. *Software Reusability. Applications and Experience*, Volume II, ACM Press, Addison-Wesley, Reading, Mass.
- Biggerstaff, T. J., and C. Richter. March 1987. "Reusability Framework, Assessment, and Directions," *IEEE Trans. on Software Engr* 4(2).
- Boehm, B. W. May 1988. "A Spiral Model of Software Development and Enhancement," *Computer* 21(5), 61-72.
- Booch, G. 1987. *Software Components with Ada*, Benjamin/Cummings, Menlo Park, Calif.
- Booch, G., and L. Williams, eds. October 1987. *Proceedings of the Workshop on Software Reuse*, Rocky Mountain Inst. of Software Engineering, SEI, MCC, Software Productivity Consortium, Boulder, Colo.
- Bott, M. F., and P. J. L. Wallis. 1988. "Ada and Software Re-use," *Software Engineering Journal* 3(5), 177-83.
- Braun, C. L., J. B. Goodenough, and R. S. Eaves. April 1985. *Ada Reusability Guidelines*, U.S. Air Force ESD 3285-2-208/2.1, SofTech.

- Brooks, F. P. 1987. "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer* 20(4), 10-19.
- Bullard, C. K., D. S. Guindi, W. B. Ligon, W. M. McCracken, and S. Rugaber. March 1989. "Verification and Validation of Resuable Ada Components," pp. 31-53 in *Guidelines Document for Ada Reuse and Metrics (Draft)*, P. A. Lesslie, R. O. Chester, and M. F. Theofanos, eds., K/DSRD-54, Martin Marietta Energy Systems, Inc., Oak Ridge, Tenn., under contract to U.S. Army, AIRMICS.
- Burton, B. A., R. W. Aragon, S. A. Bailey, K. D. Koehler, and L. A. Mayes. July 1987. "The Reusable Software Library," *IEEE Software* 4(4).
- Carstensen, H. B., Jr. March 1987. "A Real Example of Reusing Ada Software," in *Proceedings of the Conference on Software Reusability and Maintainability*, The National Institute for Software Quality and Productivity, Inc., Tysons Corner, Va.
- Cavaliere, M. J. 1989. "Reusable Code at the Hartford Insurance Group" in *Software Reusability. Applications and Experience*, Volume II, ACM Press, Addison-Wesley, Reading, Mass.
- Cheatham, T. E. September 1984. "Reusability through Program Transformations," *IEEE Trans. on Software Engr* SE10(5).
- Chen, Y. F., and C. V. Ramamoorthy. October 1986. "The C Information Abstractor," in *Session of Compsac 86*, Chicago.
- Choi, B., R. A. DeMillo, W. Du, and R. Stansifer. March 1989. "Guidelines for Reusing Operational History of Ada Software Components," pp. 55-66 in *Guidelines Document for Ada Reuse and Metrics (Draft)*, P. A. Lesslie, R. O. Chester, and M. F. Theofanos, eds., K/DSRD-54, Martin Marietta Energy Systems, Inc., Oak Ridge, Tenn., under contract to U.S. Army, AIRMICS.
- Cohen, J. July 1989. "GTE Software Reuse for Information Management Systems," in *Proceedings of the Reuse in Practice Workshop*, eds. J. Baldo and C. Braun, Software Engineering Institute, Pittsburgh, Penn.
- Conn, R. February 1986. "Overview of the DoD Ada Software Repository," *Dr. Dobb's Journal*.
- Davis, A. October 1986. Session Title: "Reusability of Program Code," in *Compsac 86*, Chicago, IL.
- DeMillo, R. A., W. Du, and R. Stansifer. June 1989. "Reuse Oriented Ada Programming-A Prototype System," in *Proceedings of the Ada Reuse and Metrics Workshop*, Atlanta.
- Druffel, L., and B. Meyer, eds. April 1988. *The 10th International Conference on Software Engineering*, Singapore.
- Dusink, E. M. July 1989. "Towards a Design Philosophy for Reuse," in *Proceedings of the Reuse in Practice Workshop*, eds. J. Baldo and C. Braun, Software Engineering Institute, Pittsburgh, Penn.
- EVB. March 1987. "Creating Reusable Ada Software," EVB Software Engineering, Inc., pp. E1-E58 in *Proceedings of the Conference on Software Reusability and Maintainability*.

52 SOFTWARE REUSE GUIDELINES

- Fairley, R., S. L. Pfleeger, T. Bollinger, A. Davis, A. J. Incorvaia, and B. Springsteen. 1989. *Final Report: Incentives for Reuse of Ada Components (Volumes 1 through V)*, George Mason University, Fairfax, Va.
- Finkelstein, A. 1988. "Reuse of Formatted Requirements Specifications," *Software Engineering Journal* 3(5), 186-97.
- Fischer, G. July 1987. "Cognitive View of Reuse and Redesign," *IEEE Software* 4(4), 60-72.
- Frakes, W. B., and B. A. Nejme. January 1987. "Software Reuse through Information Retrieval," pp. 530-35 in *Proceedings of the Twentieth Hawaii International Conference on System Sciences*, Kailua-Kona, Hawaii.
- Freeman, P. 1987. *Tutorial: Software Reusability*, IEEE Computer Society.
- Fujino, K. October 1987. "Software Factory Engineering: Today and Future," pp. 262-70 in *Proceedings of The 1987 Fall Joint Computer Conference*, Dallas.
- Gagliano, R. A., M. D. Fraser, and G. S. Owen. March 1989. "Guidelines for Reusable Ada Library Tools," pp. 79-90 in *Guidelines Document for Ada Reuse and Metrics (Draft)*, P. A. Lesslie, R. O. Chester, and M. F. Theofanos, eds., K/DSRD-54, Martin Marietta Energy Systems, Inc., Oak Ridge, Tenn., under contract to U.S. Army, AIRMICS.
- Gautier, B. 1988. "Book Review: Software Components with Ada (by Grady Booch)," *Software Engineering Journal* 3(5), 184-85.
- Geary, K. 1988. "The Practicalities of Introducing Large-Scale Software Re-use," *Software Engineering Journal* 3(5), 172-76.
- Goguen, J. A. February 1986. "Reusing and Interconnecting Software Components," *IEEE Computer* 19(2), 16-28.
- Grabow, P. C. October 1985. "Software Reuse: Where Are We Going?," p. 202 in *Conference Session, documented in Proceedings of Compsac 85*, Chicago.
- Guerrieri, E. March 1988. "Searching for Reusable Software Components with the Rapid Center Library System," pp. 395-406 in *Ancoat 1988 Proceedings of the Sixth National Conference on Ada Technology*, Arlington, Va.
- Hall, P. A. V. January 1987a. "Software Components and Reuse-Getting More Out of Your Code," *Information and Software Technology* 29(1), 38-43.
- Hall, P. A. V. December 1987b. "Software Components and Reuse," *Computer Bulletin*, , 14-15, 20.
- Hall, P. A. V. 1988. "Software Components and Reuse," a special section in *Software Engineering Journal* 3(5), 171.
- Hartman, D. February 1989. "Rapid Prototyping Using Reuse," in *Reuse and the Software Revolution Symposium*, Falcon Air Force Base, Colo.
- Hocking, D. E. March 1988. "The Next Level of Reuse," pp. 407-10 in *Ancoat 1988 Proceedings of the Sixth National Conference on Ada Technology*, Arlington, Va.
- Holibaugh, R. October 1987. "Overcoming Some Impediments to Software Reuse," in *Proceedings of the Workshop on Software Reuse*, eds. G. Booch and L.

- Williams, Rocky Mountain Inst. of Software Engineering, SEI, MCC, Software Productivity Consortium, Boulder, Colo.
- Holibaugh, R. July 1989. "Where to Begin and Why," in *Proceedings of the Reuse in Practice Workshop*, eds. J. Baldo and C. Braun, Software Engineering Institute, Pittsburgh, Penn.
- Hooper, J. W. July 1988. "Simulation Model Reuse: Issues and Approaches," pp. 51-56 in *Proceedings of The 1988 Summer Computer Simulation Conference*, Seattle.
- Hooper, J. W. April 1989. *A Perspective of Software Reuse*, ASQBG-I-89-025, U.S. Army AIRMICS.
- Horowitz, E., and J. B. Munson. September 1984. "An Expansive View of Reusable Software," *IEEE Trans. on Software Engr* SE10(5), 477-87.
- Huang, C. 1985. "Reusable Software Implementation Technology: A Review of Current Practices," p. 207 in *Proceedings of Compasac 85*.
- Humphrey, W. S. June 1989. "The Software Engineering Process: Definition and Scope," *Software Engineering Notes* 14(4), 82-83, Proceedings Publication.
- Humphrey, W. S. et al. September 1987. "A Method for Assessing the Software Engineering Capability of Contractors," CMU/SEI-87-TR-23, Software Engineering Institute, Pittsburgh.
- Hutchinson, J. W., and P. G. Hindley. 1988. "A Preliminary Study of Large-Scale Software Reuse," *Software Engineering Journal* 3(5), 208-12.
- Iscoe, N. October 1987. "A Knowledge Based and Object-Oriented Approach to Reusability Within Application Domains," in *Proceedings of the Workshop on Software Reuse*, eds. G. Booch and L. Williams, Rocky Mountain Inst. of Software Engineering, SEI, MCC, Software Productivity Consortium, Boulder, Colo.
- ISEC (U.S. Army Information Systems Engineering Command). December 1985. *ISEC Reusability Guidelines*, U.S. Army Information Systems Engineering Command 3285-4-247/2, Softech Inc., Waltham, Mass.
- Joiner, H. F. July 1989. "Position Paper on Software Reuse," in *Proceedings of the Reuse in Practice Workshop*, eds. J. Baldo and C. Braun, Software Engineering Institute, Pittsburgh, Penn.
- Jones, A., R. E. Bozeman, and W. McIver. March 1989. "A Framework for Library and Configuration Management," pp. 63-78 in *Guidelines Document for Ada Reuse and Metrics (Draft)*, P. A. Lesslie, R. O. Chester, and M. F. Theofanos, eds., K/DSRD-54, Martin Marietta Energy Systems, Inc., Oak Ridge, Tenn., under contract to U.S. Army, AIRMICS.
- Jones, G. October 1987. "Methodology/Environment Support for Reusability," in *Proceedings of the Workshop on Software Reuse*, eds. G. Booch and L. Williams, Rocky Mountain Inst. of Software Engineering, SEI, MCC, Software Productivity Consortium, Boulder, Colo.
- Jones, T. C. September 1984. "Reusability in Programming: A Survey of the State of the Art," *IEEE Trans. on Software Engr* SE10(5), 488-94.
- Kaiser, G. E., and D. Garlan. July 1987. "Melding Software Systems from Reusable Building Blocks," *IEEE Software* 4(4), 17-24.

- Kang, K. C. October 1987. "A Reuse-Based Software Development Methodology," in *Proceedings of the Workshop on Software Reuse*, eds. G. Booch and L. Williams, Rocky Mountain Inst. of Software Engineering, SEI, MCC, Software Productivity Consortium, Boulder, Colo.
- Kang, K. C. July 1989. "Features Analysis: An Approach to Domain Analysis," in *Proceedings of the Reuse in Practice Workshop*, eds. J. Baldo and C. Braun, Software Engineering Institute, Pittsburgh, Penn.
- King, R. March 1989. "Object-Oriented Data Base Modeling and Software Environments," pp. 91-94 in *Guidelines Document for Ada Reuse and Metrics (Draft)*, P. A. Lesslie, R. O. Chester, and M. F. Theofanos, eds., K/DSRD-54, Martin Marietta Energy Systems, Inc., Oak Ridge, Tenn., under contract to U.S. Army, AIRMICS.
- Kitaoka, B. J. July 1989. "Managing Large Repositories for Reuse," in *Proceedings of the Reuse in Practice Workshop*, eds. J. Baldo and C. Braun, Software Engineering Institute, Pittsburgh, Penn.
- Lanergan, R. G., and C. A. Grasso. September 1984. "Software Engineering with Reusable Design and Code," *IEEE Trans. on Software Engr* SE10(5), 498-501.
- Lee, K. J., and M. Rissman. July 1989. "Application of Domain-Specific Software Architectures to Aircraft Flight Simulators and Training Devices," in *Proceedings of the Reuse in Practice Workshop*, eds. J. Baldo and C. Braun, Software Engineering Institute, Pittsburgh, Penn.
- Lesslie, P. A., R. O. Chester, and M. F. Theofanos. March 1989. *Guidelines Document for Ada Reuse and Metrics (Draft)*, K/DSRD-54, Martin Marietta Energy Systems, Inc., Oak Ridge, Tenn., under contract to U.S. Army, AIRMICS.
- Lieberherr, K. J., and A. J. Riel. April 1988. "Demeter: A Case Study of Software Growth through Parameterized Classes," pp. 254-64 in *The 10th International Conference on Software Engineering*, Singapore.
- Lubars, M. D. October 1987. "Wide-Spectrum Support for Software Reusability," in *Proceedings of the Workshop on Software Reuse*, eds. G. Booch and L. Williams, Rocky Mountain Inst. of Software Engineering, SEI, MCC, Software Productivity Consortium, Boulder, Colo.
- Machida, S. 1985. "Approaches to Software Reusability in Telecommunications Software System," p. 206 in *Proceedings of Compsac 85*.
- Matsumoto, Y. 1989. "Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels" in *Software Reusability. Applications and Experience*, Volume II, ACM Press, Addison-Wesley, Reading, Mass.
- McKay, C. W. March 1989. "Conceptual and Implementation Models," pp. 111-48 in *Guidelines Document for Ada Reuse and Metrics (Draft)*, P. A. Lesslie, R. O. Chester, and M. F. Theofanos, eds., K/DSRD-54, Martin Marietta Energy Systems, Inc., Oak Ridge, Tenn., under contract to U.S. Army, AIRMICS.
- McNicholl, D. G., et al. 1986. *Common Ada Missile Packages (CAMP), Vol I: Overview and Commonality Study Results*, AFATL-TR-85-93, McDonnell Douglas, St. Louis.

- Meyer, B. March 1987. "Reusability: The Case for Object-Oriented Design," *IEEE Software* 4(2), 50-64.
- Mittermeir, R. T., and W. Rossak. October 1987. "Software Bases and Software Archives: Alternatives to Support Software Reuse," pp. 21-28 in *Proceedings of The 1987 Fall Joint Computer Conference*, Dallas.
- Murine, G. E. March 1987. "Recent Japanese Advances in Reusability and Maintainability," pp. 11-115 in *Proceedings of the Conference on Software Reusability and Maintainability*, ed. E. Yourdon, The National Institute for Software Quality and Productivity, Inc., Tysons Corner, Va.
- Neighbors, J. M. September 1984. "The Draco Approach to Constructing Software from Reusable Components," *IEEE Trans. on Software Engr* SE10(5), 564-74.
- Neighbors, J. M. October 1987. "Report on the Domain Analysis Working Group Session," in *Proceedings of the Workshop on Software Reuse*, eds. G. Booch and L. Williams, Rocky Mountain Inst. of Software Engineering, SEI, MCC, Software Productivity Consortium, Boulder, Colo.
- Nissen, J., and P. Wallis. 1984. *Portability and Style in Ada*, Cambridge University Press.
- Notkin, D., and W. G. Griswold. April 1988. "Extension and Software Development," pp. 274-83 in *The 10th International Conference on Software Engineering*, Singapore.
- Onuegbue, E. O. January 1987. "Software Classification as an Aid to Reuse: Initial Use as Part of a Rapid Prototyping System," pp. 521-29 in *Proceedings of the Twentieth Hawaii International Conference on System Sciences*, Kailua-Kona, Hawaii.
- Oskarsson, O. 1989. "Reusability of Modules with Strictly Local Data and Devices—A Case Study" in *Software Reusability. Applications and Experience*, Volume II, ACM Press, Addison-Wesley, Reading, Mass.
- Palmer, C. July 1989. "Reuse in Practice," in *Proceedings of the Reuse in Practice Workshop*, eds. J. Baldo and C. Braun, Software Engineering Institute, Pittsburgh, Penn.
- Perry, J. M., and M. Shaw. July 1989. "The Role of Domain Independence in Promoting Software Reuse," in *Proceedings of the Reuse in Practice Workshop*, eds. J. Baldo and C. Braun, Software Engineering Institute, Pittsburgh, Penn.
- Plinta, C. July 1989. "A Model Solution for the C³I Domain," in *Proceedings of the Reuse in Practice Workshop*, eds. J. Baldo and C. Braun, Software Engineering Institute, Pittsburgh, Penn.
- Presson, P. E., J. Tsai, T. P. Bowen, J. V. Post, and R. Schmidt. July 1983. *Software Interoperability and Reusability Guidebook for Software Quality Measurement, Volumes I and II*, Boeing Aerospace Co.
- Prieto-Diaz, R. October 1987a. "Domain Analysis for Reusability," pp. 23-29 in *Proceedings of Compsac 87*, Tokyo.
- Prieto-Diaz, R. October 1987b. "Faceted Classification and Reuse Across Domains," in *Proceedings of the Workshop on Software Reuse*, eds. G. Booch

- and L. Williams, Rocky Mountain Inst. of Software Engineering, SEI, MCC, Software Productivity Consortium, Boulder, Colo.
- Prieto-Diaz, R., and P. Freeman. January 1987. "Classifying Software for Reusability," *IEEE Software* 4(1), 6-16.
- Prywes, N. S., and E. D. Lock. 1989. "Use of the Model Equational Language and Program Generator by Management Professionals" in *Software Reusability. Applications and Experience*, Volume II, ACM Press, Addison-Wesley, Reading, Mass.
- Pyster, A., and B. Barnes. December 1987. *The Software Productivity Consortium Reuse Program*, SPC-TN-87-016, Software Productivity Consortium, Reston, Va.
- Ramamoorthy, C. V., V. Garg, and A. Prakash. October 1986. "Support for Reusability in Genesis," pp. 299-305 in *Session of Compsac 86*, Chicago.
- Ramamoorthy, C. V., and R. T. Yeh, eds. October 1987. *Proceedings of The 1987 Fall Joint Computer Conference*, Dallas.
- Rogerson, A. M., and S. C. Bailin. July 1987. "Software Reusability Environment Prototype: Experimental Approach," in *In Proceedings of The Tenth Minnowbrook Workshop*, Software Reuse, Blue Mountain Lake, N.Y.
- Sayani, H. March 1987. "Applications in Reverse Software Engineering," pp. L1-L15 in *Proceedings of the Conference on Software Reusability and Maintainability*, ed. E. Yourdon, The National Institute for Software Quality and Productivity, Inc., Tysons Corner, Va.
- Selby, R. W. 1989. "Quantitative Studies of Software Reuse," in *Software Reusability: Vol. II Applications and Experience*, eds. T. J. Biggerstaff and A. J. Perlis, pp. 213-233.
- Shriver, B. D., and R. H. Sprague, Jr., eds. January 1987. *Proceedings of the Twentieth Hawaii International Conference on System Sciences*, Kailua-Kona, Hawaii.
- Simos, M. A. October 1987. "The Domain-Oriented Software Life Cycle: Towards an Extended Process Model for Reusability," in *Proceedings of the Workshop on Software Reuse*, eds. G. Booch and L. Williams, Rocky Mountain Inst. of Software Engineering, SEI, MCC, Software Productivity Consortium, Boulder, Colo.
- Smith, S. R. 1989. "An Advanced Full-Text Information Retrieval System," Ph.D. dissertation, The University of Alabama, Huntsville, Ala.
- Solderitsch, J. July 1989. "The Reusability Library Framework: Working Toward an Organon," in *Proceedings of the Reuse in Practice Workshop*, eds. J. Baldo and C. Braun, Software Engineering Institute, Pittsburgh, Penn.
- St. Dennis, R. J. May 1986. *A Guidebook for Writing Reusable Source Code in Ada (R)*, Version 1.1, CSC-86-3:8213, Honeywell, Golden Valley, Minn.
- St. Dennis, R. J. January 1987. "Reusable Ada Software Guidelines," pp. 513-20 in *Proceedings of the Twentieth Hawaii International Conference on System Sciences*, Kailua-Kona, Hawaii.
- Standish, T. A. September 1984. "An Essay on Software Reuse," *IEEE Trans. on Software Engr* SE10(5), 494-97.

- Tarumi, H., K. Agusa, and Y. Ohno. April 1988. "A Programming Environment Supporting Reuse of Object-Oriented Software," pp. 265-73 in *The 10th International Conference on Software Engineering*, Singapore.
- Taylor, C. July 1989. "Software Reuse," in *Proceedings of the Reuse in Practice Workshop*, eds. J. Baldo and C. Braun, Software Engineering Institute, Pittsburgh, Penn.
- Tracz, W. January 1987a. "RECIPE: A Reusable Software Paradigm," pp. 546-55 in *Proceedings of the Twentieth Hawaii International Conference on System Sciences*, eds. B. D. Shriver and R. H. Sprague, Jr., Kailua-Kona, Hawaii.
- Tracz, W. July 1987b. "Making Reuse a Reality," *IEEE Software* 4(4).
- Tracz, W. October 1987c. "Software Reuse Myths," in *Proceedings of the Workshop on Software Reuse*, eds. G. Booch and L. Williams, Rocky Mountain Inst. of Software Engineering, SEI, MCC, Software Productivity Consortium, Boulder, Colo.
- Tracz, W. 1988a. *Tutorial: Software Reuse: Emerging Technology*, IEEE Computer Society.
- Tracz, W. 1988b. "Ada Reusability Efforts: A Survey of the State of the Practice," pp. 23-32 in *Tutorial: Software Reuse: Emerging Technology*, IEEE Computer Society.
- Tully, C., ed. June 1989. "Proceedings of the 4th International Software Process Workshop, Devon UK, May, 1988," *Software Engineering Notes* 14(4), 82-83.
- Utter, D. F. 1985. "Reusable Software Requirements Documents," *Conference Session, documented in Proceedings of Compsac 85*, 204.
- Van Scoy, R., and C. Plinta. July 1989. "An Informal Experiment in Reuse," in *Proceedings of the Reuse in Practice Workshop*, eds. J. Baldo and C. Braun, Software Engineering Institute, Pittsburgh, Penn.
- Vogelsong, T. July 1989. "Reusable Ada Packages for Information System Development (RAPID)—An operational Center of Excellence for Software Reuse," in *Proceedings of the Reuse in Practice Workshop*, eds. J. Baldo and C. Braun, Software Engineering Institute, Pittsburgh, Penn.
- Wald, E. 1986. *STARS Reusability Guidebook, V4.0*, Department of Defense STARS, Draft.
- Wong, W. 1986. *A Management Overview of Software Reuse*, 500-142, National Bureau of Standards, Washington, D.C.
- Wood, M., and I. Sommerville. 1988. "An Information Retrieval System for Software Components," *Software Engineering Journal* 3(5), 198-207.
- Yamamoto, S., and S. Isoda. October 1986. "SOFTDA—A Reuse-Oriented Software Design System," pp. 284-90 in *Session of Compsac 86*, Chicago.
- Yeh, R. T., and T. A. Welch. October 1987. "Software Evolution: Forging a Paradigm," pp. 10-12 in *Proceedings of The 1987 Fall Joint Computer Conference*, Dallas.
- Yourdon, E., ed. March 1987. *Proceedings of the Conference on Software Reusability and Maintainability*, The National Institute for Software Quality and Productivity, Inc., Tysons Corner, Va.

APPENDIX A: COLLECTED GUIDELINES

CONTENTS

A.1 MANAGERIAL GUIDELINES	60
A.1.1 Management Issues	60
A.1.1.1 Leadership and Organizational Structure	60
A.1.1.2 Organizational Behavior	61
A.1.1.3 Contractual and Legal Considerations	61
A.1.1.4 Financial Considerations	61
A.1.2 Software Engineering Process Incorporating Reuse	61
A.2 TECHNICAL GUIDELINES (RELATIVE TO THE SOFTWARE LIFE CYCLE)	62
A.2.1 Guidelines Spanning the Life Cycle	62
A.2.1.1 General Guidelines	62
A.2.1.2 Component Quality	62
A.2.1.3 Classifying and Storing Components	63
A.2.1.4 Searching and Retrieving	63
A.2.1.5 Understanding and Assessing Components	63
A.2.1.6 Adapting Components	63
A.2.2 Domain Analysis	64
A.2.3 Requirements Determination	64
A.2.4 High-Level Design	64
A.2.5 Detailed Design	64
A.2.6 Coding and Unit Testing	65
A.2.6.1 Subprograms	65
A.2.6.2 Packages	66
A.2.6.3 Visibility Rules	66
A.2.6.4 Tasks	66
A.2.6.5 Program Structure and Compilation Issues	67
A.2.6.6 Generic Units	68
A.2.6.7 Exceptions	69
A.2.7 Integration Testing	69
A.2.8 Maintenance	69
A.3 GETTING STARTED (STEPS)	69

The following is a complete listing of the software reuse guidelines presented in this document. An attempt has been made to arrange the guidelines in the logical order for operational implementation. Following each guideline is a page number in square brackets. This is the page in the body of this document that first presents the guideline.

Each guideline has been given a unique identifier, for example LS1, to simplify referring to each guideline. LS1 through LS13 are the guidelines pertaining to corporate leadership and organizational structure. OB1 and OB2 relate to organizational behavior. CL1 through CL5 are contractual and legal guidelines. F1 through F4 are financial guidelines. SP1 through SP4 are the software process guidelines. DA1

through DA5 are domain analysis guidelines. CC1 through CC21 are component creation guidelines. Guidelines for Ada subprograms, packages, visibility, tasks, program structure, generic units, and exceptions are labeled AS1 through AS12, AP1 through AP6, AV1 through AV4, AT1 through AT16, AP1 through AP9, AG1 through AG15, and AE1 through AE4, respectively. Guidelines for reusing components are labeled RC1 through RC11. Quality guidelines are labeled Q1 through Q4. Verification and validation guidelines are labeled V1 through V8.

Guidelines are introduced in the text as the discussion develops the rationale and background for the guidelines. As a result, guidelines in a top level category are not always listed sequentially under one heading. For example, CC18 through CC21 are component creation guidelines that relate to classifying and storing components and are therefore listed in Section A.2.1.3, Classifying and Storing Components. We do not feel this detracts from the utility of the guideline identifiers, but enhances their utility by indicating the category of guideline that addresses the section topic.

A.1 MANAGERIAL GUIDELINES

A.1.1 Management Issues

A.1.1.1 Leadership and Organizational Structure

- LS1: *Upper-level management must set reuse goals, create an organizational infrastructure to support software reuse, establish policies, and provide necessary resources. [p. 8]*
- LS2: *Mid-level management must develop procedures, populate the organizational structure, allocate resources, and establish controls and metrics to achieve goals. [p. 8]*
- LS3: *Project-level management and technical personnel must carry out the reuse/reusability activities in individual projects. [p. 8]*
- LS4: *Establish an organizational entity whose charter is to promote reuse considerations at the corporate level. [p. 8]*
- LS5: *Evaluate the suitability of establishing the reuse matrix structure and domain reuse groups. [p. 8]*
- LS6: *Structure software development by domains amenable to reuse considerations. [p. 8]*
- LS7: *Establish strong connections between reuse and maintenance activities. [p. 8]*
- LS8: *Provide different types of training for managers, developers, and domain reuse specialists. [p. 8]*
- LS9: *Make personnel assignments that take reuse and reusability into account. [p. 8]*
- LS10: *Assign reuse facilitators to development groups. [p. 8]*
- LS11: *Allow two to three years after initiating software reuse before expecting economic advantages from the program. [p. 8]*
- LS12: *Provide a corporate financial "safety net" for projects practicing reuse; provide funding for generation of reusable components. [p. 8]*

LS13: *Managers in federal organizations must take the initiative to influence the adoption of reuse within their own organizations and within contracts they direct. [p. 8]*

A.1.1.2 Organizational Behavior

OB1: *Provide incentive rewards to participate in reuse. [p. 9]*

OB2: *Seek to enhance psychological job satisfaction to motivate willing participation in reuse. [p. 9]*

A.1.1.3 Contractual and Legal Considerations

CL1: *Seek contractual means to require or to encourage contractors to create reusable software (from which the government gains future benefits) and to reuse existing software (which gives the government immediate benefits). [p. 11]*

CL2: *Establish and enforce reuse practices within government development groups. [p. 11]*

CL3: *Require reuse within a group of related contracts (e.g., by a prime contractor and subcontractors). [p. 11]*

CL4: *Seek means to alter project funding approaches to encourage creation of reusable software. [p. 11]*

CL5: *Seek resolution to the legal issues of potential liability and partial ownership. [p. 11]*

A.1.1.4 Financial Considerations

F1: *Establish mechanisms to accumulate an organizational database of historical financial data relative to software production and maintenance, including reuse activities. [p. 11]*

F2: *Provide cost modeling tools, to the extent feasible, in concert with organizational data for reuse/reusability decision assessments (including make versus reuse versus buy decisions). [p. 11]*

F3: *Consider/model costs over multiple projects. [p. 12]*

F4: *Establish mechanisms to share the cost of developing reusable components across multiple projects. [p. 12]*

A.1.2 Software Engineering Process Incorporating Reuse

SP1: *Initiate action to establish a software engineering process (including development and maintenance) that includes reuse and reusability as important, integral, natural, and inescapable elements. [p. 16]*

SP2: *Augment DOD-STD-2167A with refinements that specifically support and encourage reusability and reuse (e.g., the five-step generic reuse/reusability model). [p. 16]*

SP3: *Consider the object-oriented methodology for use within the software process. [p. 16]*

62 SOFTWARE REUSE GUIDELINES

SP4: *Automate activities within the software process as understanding and experience permit. [p. 16]*

A.2 TECHNICAL GUIDELINES (RELATIVE TO THE SOFTWARE LIFE CYCLE)

A.2.1 Guidelines Spanning the Life Cycle

A.2.1.1 General Guidelines

- CC1: *Provide domain analysis results within the reuse framework—explicitly and/or implicitly. [p. 23]*
- CC2: *Make careful assessments, including financial predictions, in deciding whether to develop a reusable component. [p. 23]*
- CC3: *Prepare for reuse all more-abstract life cycle representations of a reusable component (e.g., prepare requirements specification for a high-level design). [p. 23]*
- CC4: *Record and supply adaptation suggestions with a reusable component. [p. 23]*
- CC5: *Generalize a reusable component to the extent practical during its preparation. [p. 23]*

A.2.1.2 Component Quality

- Q1: *Set standards to be met by all library components. [p. 33]*
- Q2: *Emphasize stringent V&V for reusable components, stressing portability and adaptability. [p. 33]*
- Q3: *Emphasize enforcement of standards and practices by the quality assurance group; employ a reusability checklist. [p. 33]*
- Q4: *Establish and operate an effective configuration management program for the reuse library. [p. 34]*
- V1: *State environment compatibility explicitly in the requirements specification. [p. 34]*
- V2: *Specify constraints on the use of reusable components as assertions; include assertions within the component specification and (if practical) within the executable code. [p. 34]*
- V3: *Construct code for portability and adaptability (rather than attempting to isolate inadequacies by testing). [p. 34]*
- V4: *Parameterize specifications that are dependent on the machine environment so that the behavior of the component is expressed relative to a part of the machine environment. [p. 34]*
- V5: *Classify reusable components along each of the dimensions of concurrency, space utilization, space reclamation, and iterator availability; make the classification a part of the component specification. [p. 34]*

- V6: *Use suggested classification of reuse errors to recognize or to detect errors automatically; remove errors, if possible, or adequately document their existence for location and modification when necessary. [p. 34]*
- V7: *Employ a comprehensive testing method, consisting of a combination of approaches, to detect reuse errors (including simulation of the execution environment, static analysis, mutation analysis, and constraint-based analysis). [p. 34]*
- V8: *Include adequacy criteria that reduce the amount of testing needed and that provide a measure of a test suite's effectiveness. [p. 34]*

A.2.1.3 Classifying and Storing Components

- CC18: *Determine approach(es) for classifying and storing components based, e.g., on domain analysis. [p. 37]*
- CC19: *Represent relationships between a component and its more (and less) abstract representations (as to life-cycle phase). [p. 37]*
- CC20: *Represent relationships between a component and others that may collectively solve a given problem (or class of problems). [p. 37]*
- CC21: *Document each component thoroughly on-line, including user documentation and programming (i.e., maintenance) documentation. [p. 37]*

A.2.1.4 Searching and Retrieving

- RC1: *Devise and implement a mechanism for search and retrieval supporting query and browsing modes. [p. 39]*
- RC2: *Emphasize user-friendly interface for search and retrieval, [p. 39]*
- RC3: *Provide indication of "goodness of match" of components to a query. [p. 39]*

A.2.1.5 Understanding and Assessing Components

- RC4: *Seek to understand reusable components chiefly through effective domain analysis, good software development practices, and good classification and storage mechanisms. [p. 40]*
- RC5: *Seek approaches/tools to help understand software not specifically prepared for reuse (e.g., reverse engineering). [p. 40]*
- RC6: *Use operational history of components in assessing their suitability for reuse. [p. 40]*
- RC7: *Obtain feedback from users of components, including number of uses, degree of satisfaction, and errors. [p. 40]*

A.2.1.6 Adapting Components

- RC8: *Use higher-abstraction representations in adapting a component (e.g., use design when adapting code). [p. 42]*
- RC9: *Emphasize the use of available metrics/tools to assess adaptation effort. [p. 42]*

A.2.2 Domain Analysis

- DA1: *Select domain(s) carefully for analysis, based on the maturity and stability of the organization's activities within each domain and on the planned emphasis the domain is to receive. [p. 21]*
- DA2: *Determine and apply a systematic approach to domain analysis, yielding a domain model, a set of domain terminology, and a domain architecture. [p. 21]*
- DA3: *Use domain analysis results as a basis for classifying, storing, and retrieving reusable components. [p. 21]*
- DA4: *Use domain analysis results as a basis for decisions about the advisability of investing in specific instances of reusable software. [p. 21]*
- DA5: *Use domain analysis results to help understand how existing domain-specific reusable software may be applied. [p. 21]*

A.2.3 Requirements Determination

- CC6: *State as a requirement the reuse of software and/or the creation of reusable software. [p. 23]*
- CC7: *During the requirements determination phase, conduct the following activities:*
 - 1. *Make use of available requirements components as appropriate.*
 - 2. *Structure requirements to take advantage of available high-level designs.*
 - 3. *Emphasize preparation of reusable requirements components (from newly developed requirements and modified, previously available requirements). [p. 23]*

A.2.4 High-Level Design

- CC8: *During the high-level design phase, conduct the following activities:*
 - 1. *Make use of available high-level-design components as appropriate.*
 - 2. *Structure the high-level design to take advantage of available detailed designs.*
 - 3. *Emphasize preparation of reusable high-level-design components (from newly developed high-level designs and modified, previously available high-level designs). [p. 23]*

A.2.5 Detailed Design

- CC9: *During the detailed-design phase, conduct the following activities:*
 - 1. *Make use of available detailed-design components as appropriate.*
 - 2. *Structure the detailed design to take advantage of available code modules.*
 - 3. *Emphasize preparation of reusable detailed-design components (from newly developed detailed designs and modified, previously available detailed designs). [p. 23]*

A.2.6 Coding and Unit Testing

- CC10: *During the coding and unit-testing phase, conduct the following activities:*
1. *Make use of available code components and test cases as appropriate.*
 2. *Emphasize preparation of reusable code components and test cases (from newly developed code/test cases and modified, previously available code/test cases). [p. 23]*
- CC13: *Supply reusable code in the form of a generator or a transformation system for greater reuse leverage, when practical. [p. 26]*
- CC14: *In preparing code blocks, use Ada generics, parameterized procedures, and code templates for greater reuse generality, as appropriate. [p. 26]*
- CC15: *Emphasize good programming style in developing reusable code, creating code exhibiting understandability, reliability, and maintainability. [p. 26]*
- CC16: *Emphasize Ada as a suitable programming language for reusable code. [p. 31]*
- CC17: *Establish a set of organizational guidelines for code development. [p. 31]*
- RC10: *Use existing mechanisms for composition to the extent possible (e.g., procedure linking, UNIX pipes, inheritance in object-oriented languages, etc.). [p. 43]*
- RC11: *Seek automated approaches to composition as understanding permits. [p. 43]*

A.2.6.1 Subprograms

- AS1: *Separate subprogram declarations and bodies for ease of recompilation and modification. [p. 27]*
- AS2: *All reusable subprograms except a main program must be written within a library unit package. [p. 27]*
- AS3: *Use subprogram declarations to specify interfaces to reusable objects. Use subprogram bodies to implement these interfaces and properties of the objects. [p. 27]*
- AS4: *Write subprogram interfaces at an appropriate abstract level. [p. 27]*
- AS5: *Standardize the format for first-level, package-nested subprogram declarations, including regions for purpose, parameter descriptions, and associated documentation. [p. 27]*
- AS6: *Standardize the format for secondary unit (subunit) and first-level, package-body-nested subprogram bodies, including regions for revision history, purpose, associated documentation, parameter description, assumptions/resources required, side effects, diagnostics, data declarations, packages, operations, and algorithmic code. [p. 27]*
- AS7: *Write subprogram bodies to handle interaction with/effects on their environment. [p. 27]*

- AS8: *Write subprogram bodies with one normal exit and a grouped set of abnormal exits via exception handlers. [p. 27]*
- AS9: *Write subprogram bodies that pass results back to callers rather than use results to effect their function. [p. 27]*
- AS10: *Exploit formal parameter modes to clarify subprogram interface semantics. [p. 27]*
- AS11: *Use named-parameter associations for calls on subprograms with more than three parameters or in any case for interface clarity. [p. 27]*
- AS12: *Minimize subprogram overloading. [p. 28]*

A.2.6.2 Packages

- AP1: *Write library unit package specifications and bodies in separate files for ease of recompilation and modification. [p. 28]*
- AP2: *Use package specifications to specify the interface to object abstractions; use package bodies to encapsulate implementation-specific details of these abstractions not needed by client software. [p. 28]*
- AP3: *Design packages to implement interfaces to reusable objects at a consistent abstract level. [p. 28]*
- AP4: *Standardize the format for library unit package specifications, including various regions for revision history, purpose, associated documentation, diagnostics, packages, data declarations, operations, and private types. [p. 28]*
- AP5: *Standardize the format for secondary unit package bodies, including regions for revision history, purpose, associated documentation, assumptions/resources required, side effects, diagnostics, packages, data declarations, and operations and initialization code. [p. 28]*
- AP6: *Use private or limited private types and the private part of package specifications to restrict client software's view of data and operations on that data. [p. 28]*

A.2.6.3 Visibility Rules

- AV1: *Do not use USE context clauses. [p. 28]*
- AV2: *Use renaming declarations to resolve name conflicts with the environment. [p. 28]*
- AV3: *Use renaming declarations to facilitate modifying reusable software to represent new object abstractions. [p. 28]*
- AV4: *Do not hide package standard (i.e., do not use the identifier STANDARD as a user-defined name). [p. 28]*

A.2.6.4 Tasks

- AT1: *Separate task declarations and bodies for ease of recompilation and modification. [p. 28]*

- AT2: *Use task declarations to specify interfaces to reusable objects. Use task bodies to implement these interfaces and properties of the objects. [p. 28]*
- AT3: *Write task interfaces at an appropriate abstract level. [p. 28]*
- AT4: *Standardize the format for first-level, package- nested-task declarations, including regions for purpose, entry descriptions, representation clause descriptions, and associated documentation. [p. 28]*
- AT5: *Standardize the format for secondary unit (subunit) and first-level, package-body-nested-task bodies, including regions for revision history, purpose, associated documentation, assumptions/resources required, side effects, diagnostics, packages, data declarations, and operations and algorithmic code. [p. 28]*
- AT6: *Write task bodies to effectively handle interaction with/effects on their environment. [p. 29]*
- AT7: *Write task bodies with one normal exit or termination point and a grouped set of abnormal exits via exception handlers. [p. 29]*
- AT8: *Write task body accept statements that pass results back to callers of the task rather than use results to effect task function. [p. 29]*
- AT9: *Use task types to define reusable operations on data and task objects to implement particular (distinct) instances of these operations. [p. 29]*
- AT10: *Exploit entry formal parameter modes to clarify task interface semantics. [p. 29]*
- AT11: *Group all default parameters in entry parameter specifications at the end of the specifications. [p. 29]*
- AT12: *Use named parameter associations for calls to task entries with greater than three parameters or in any case for interface clarity. [p. 29]*
- AT13: *Minimize entry overloading. [p. 29]*
- AT14: *Write all select statements with an else part or include a handler for the PROGRAM_ERROR exception at the end of the enclosing task block. [p. 29]*
- AT15: *Minimize use of task priorities or modify priorities accordingly when composing tasks with other tasks for the sake of reuse. [p. 29]*
- AT16: *Minimize use of abort statements. [p. 29]*

A.2.6.5 Program Structure and Compilation Issues

- APS1: *Use library unit package specifications as the encapsulation mechanisms for directly reusable software (i.e., data and operations on the data). [p. 29]*
- APS2: *Allow only first-level, nested-nonpackage entities in library unit package specifications to form the basis for catalogued, directly reusable objects/software. [p. 29]*
- APS3: *Use secondary unit package bodies, package specifications containing only data, and subunits corresponding to first-level, package-body-nested stubs as the encapsulation mechanisms for indirectly reusable software. [p. 29]*

- APS4: Use *WITH* clauses on package specifications to reference only data needed in specifications. *WITH* clauses can be used freely on package bodies as needed. [p. 29]
- APS5: Use subunits to achieve modularity and ease of recompilation. [p. 29]
- APS6: Do not split an abstraction across several packages. Use layers of packages to reflect the abstraction. [p. 29]
- APS7: Use separate compilation and separate specifications and bodies to achieve modularity and ease of recompilation. [p. 29]
- APS8: Implement bodies of subprograms declared in a package specification as subunits of the package body. [p. 29]
- APS9: Minimize use of *Pragma Elaborate*. [p. 30]

A.2.6.6 Generic Units

- AG1: Use generic program units (i.e., packages and subprograms) to parameterize reusable software parts. [p. 30]
- AG2: Use generic program units to specify precise module interfaces/imports and exports. [p. 30]
- AG3: Use generics to allow specification of multiple instances of reusable software as compared to reuse of one shared instance. [p. 30]
- AG4: Use base types rather than subtypes to specify the type of a generic formal object or generic formal subprogram parameter or to specify result types. [p. 30]
- AG5: Standardize the format for *Library* unit and first-level, package-nested, generic unit declarations, including a region for description of generic parameters as well as standard information required for nongeneric subprogram and package declarations. [p. 30]
- AG6: Separate generic declarations from bodies for ease of recompilation and modification. [p. 30]
- AG7: Exploit generic formal object parameter modes to clarify interface semantics. [p. 30]
- AG8: Use generic type definitions to clarify interface semantics and module operation. [p. 30]
- AG9: Use additional generic parameters as necessary to effect inheritance of desired operators on generic formal types. [p. 30]
- AG10: Minimize generic formal subprogram parameter overloading and overloading of subprograms in generic packages. [p. 30]
- AG11: Minimize use of the box notation (*is <>*) to specify default generic formal subprograms as parameters. [p. 30]
- AG12: Use basic operations/attributes associated with generic formal types to provide required generality to generic bodies. [p. 30]
- AG13: Use named-parameter association in actual parameter parts of generic instantiations. [p. 30]

AG14: *Use default parameters for generic, actual parameters whenever possible. [p. 30]*

AG15: *Create particular instantiations of generic units corresponding to common uses of reusable software. [p. 30]*

A.2.6.7 Exceptions

AE1: *For each assumption a subroutine depends on to operate correctly, define an exception that is to be raised when the assumption is violated. [p. 30]*

AE2: *For every situation that would raise an exception, define a function that indicates whether the exception would be raised. [p. 30]*

AE3: *Raise an exception if the user can easily fix the problem. Let the user provide a subprogram to fix the problem if raising an exception causes loss of information. [p. 30]*

AE4: *Provide a subprogram to return all available information describing the nature of an exception situation. [p. 31]*

A.2.7 Integration Testing

CC11: *During integration testing, conduct the following activities:*

1. *Make use of available test plans, test cases, and test results as appropriate.*
2. *Emphasize preparation of reusable test plans/cases/results (from newly developed tests and modified, previously available tests). [p. 23]*

A.2.8 Maintenance

CC12: *Recognizing that the maintenance phase contains as subphases the software development phases, apply to maintenance the guidelines provided above for requirements, design, coding, and testing. [p. 24]*

A.3 GETTING STARTED (STEPS)

The following steps summarize a suggested phased approach for implementing a software reuse program [p. 48]. The steps are stated in a suggested order of occurrence. Of course, there are implied feedback loops and iterations in these steps. The exact implications of each step must be determined by current status of the "bootstrapping" process. It should be noted that carrying out these steps implies the use of the managerial and technical guidelines.

1. *Make an assessment of current software engineering practices and remedy major shortcomings.*
2. *Obtain top-level management support to undertake investigation of reuse feasibility.*
3. *Identify one or more application areas (if any) that are important to the organization's mission, are well understood, and have recurring similar software requirements.*
4. *Conduct an inventory of reusable assets for the identified application area(s).*

70 *SOFTWARE REUSE GUIDELINES*

5. Establish an initial library of reusable components.
6. Determine and conduct a pilot software project employing reuse.
7. Evaluate experience/success with reuse in the pilot project, present results to management and obtain a decision whether to proceed.
8. Expand reuse activities to additional application domains and organizational segments, as success and management approval warrant.
9. Conduct the following activities as part of the practice of software reuse (in the pilot project and in all succeeding reuse efforts):
 - * Institute management policies and practices to encourage reuse.
 - * Institute/carry out a software engineering process incorporating the creation and use of reusable products.
 - * Adjust the organizational structure and staffing as appropriate to support reuse.
 - * Implement/update library mechanisms.
 - * Perform domain analysis of selected domains and develop/acquire enough reusable components to conduct reuse for the domains.
 - * Continually assess the effectiveness of the reuse-based process, and adjust/augment it as appropriate.

ACRONYMS

AFATDS	U.S. Army Advanced Field Artillery Technical Data System
AIRMICS	U.S. Army Institute for Research in Management Information, Communications and Computer Sciences
AJPO	DOD Ada Joint Program Office
ALS	Automated Library System
ARSC	Applications of Reusable Software Components
CAMP	Common Ada Missile Packages
CASE	Computer-aided software engineering tools
CECOM	U.S. Army Communications Electronics Command
DOD	Department of Defense
DOE	Department of Energy
EAP	Experimental Aircraft Programme in the United Kingdom
EI	Extension Interpreter
ERA	entity-relationship-attribute
ESF	The European Software Factory
ESL	the Entity Specification Language
FFP	Firm Fixed Price Contract
GSFC	NASA Goddard Space Flight Center
GSU	Georgia State University
IR	information retrieval
ISEC	U.S. Army Information Systems Engineering Command
LIL	library interconnection language
MCC	Microelectronics and Computer Technology Corporation
NASA	National Aeronautics and Space Administration
OOD	object-oriented design
OPTI	Department of Commerce Office of Productivity, Technology, and Innovation
RADC	U.S. Air Force Rome Air Development Center
RAPID	reusable Ada packages for information systems development
RCL	RAPID Center Library
REX	Resource Extractor
RSL	the reusable software library
SDC	U.S. Army Strategic Defense Command
SEI	Software Engineering Institute
SPC	Software Productivity Consortium
SRE	Software Reuse Environment
STARS	Software Technology for Adaptable, Reliable Systems
USM	utility systems management